MS-DOS ver3.1

# 実用マクロアセンブラ

河西朝雄 著

技術評論社

# 実用マクロアセンブラ

河西朝雄 著

技術評論社

## はじめに

前著『98FAM BOOKS CP/M-86 vs MS-DOS』から1年半ほどが経過しました．その間にどうやら16ビットパソコンのOSの主流はMS-DOSに落ち着いて来たようです．それに伴い，「パソコン＝BASICという世界」から抜け出し，パソコンのOS搭載が常識化してきたことも歓迎したいと思います．

さて，MS-DOSというOSの上では，アセンブリ言語や高級言語による多様なプログラミングが行え，さらに日本語ワードプロセッサやMultiplanなどのアプリケーションソフトも利用できます．また，本書で扱うMASMというマクロアセンブラもMS-DOSの上で動作します．

本書はMS-DOSに標準装備されているマクロアセンブラMASM(MACRO-86)を用いたアセンブリ言語のプログラミングについて説明します．

MS-DOSはインテル社のCPU 8086用のOSです．そのOSの上でアセンブリ言語プログラムを作成するには，まず8086の命令についての知識が必要になります．さらに，8086は1Mバイトのメモリ空間をセグメントという概念を用いて管理しているため，セグメントの管理やメモリ・アドレッシングなどの考え方も知っておく必要があります．これらについては第1章で説明します．

次にMS-DOSの上でアセンブリ言語プログラムを作成するためには，アセンブリ言語の標準的な書式を覚え，アセンブル／リンクの典型的なオペレーション方法を学ばなければなりません．これらについては第2章で説明します．

こうした基礎概念が身に付いたならば，簡単なプログラムを自作してみることがプログラミングの上達への早道です．このための例題を第3章に示しました．

マクロアセンブラMASMの重要な概念(セグメント／モジュール／データの定義／マクロ)と命令(擬似命令と演算子)については第4，5章で詳しく説明します．.

マクロアセンブラの支援ユーティリティ(MASM／LINK／LIB／CREF／DEBUGそしてMS-DOS ver 3.1で加わったSYMDEB)のオペレーションに関する詳しい解説は第6章に示しました(SYMDEBについては第12章)．

さらに，MS-DOS上で実用的なプログラムを作成するためには，MS-DOSの構造を知っておく必要があります．そこでPSP，FCB，FATなどの技術情報を第7章に示しました．

アセンブリ言語のプログラムを作成するには，一般にかなりの労力が要るとされています．これはコンソールやディスクへのアクセスに関する命令を自分で作らなけれ

ばならないからです．MASMのマクロ機能を使って基本的な処理をマクロ定義したヘッダファイルを作っておき，ユーザプログラムでこれを取り込んで使用すれば，プログラマの負担が著しく軽減され，労力が少なくてすみます．その上，一種の高級言語を使って記述するような感覚でプログラミングができます．このMS-DOSのファンクションコール関係のマクロを第8章に，エスケープシーケンスによる画面制御マクロを第9章に示し，これらのマクロ定義ヘッダファイルを利用したユーティリティプログラムの具体例を第10章に示しました．

さらに，MS-DOSやCP/M-86などのOSは，普通，グラフィックをサポートしていないので，PC-9801シリーズ(E／F／M／VF／VM)のグラフィックLIOを利用したグラフィック・インタフェースの例を第11章に示しました．

また，MS-DOS ver 3.1へのバージョン･アップに伴ったマクロアセンブラによる開発に関する追加補足事項を第12章に示しました．

一般にプログラムの開発効率が悪いといわれるアセンブリ言語も，MASMのマクロ機能やモジュールといった概念を導入することで，かなり使い勝手の良い言語になります．

アセンブリ言語によるプログラミングはBASICや他の高級言語のような気軽さで行うわけにはいきません．それなのに，なぜ，面倒なアセンブリ言語でプログラミングを行わなければならないのかというと，

・コンパイラやインタプリタの言語仕様外の処理をプログラミングできる

・作成されたプログラムはオブジェクト効率がよく，高速である

という他のプログラミング言語では得られない長所があるからです．

本書がマクロアセンブラの概念や活用法を理解する上での一助になれば幸いです．

最後に本書の出版に際して，御尽力いただいた技術評論社の編集部の方々に厚く御礼を申しあげます．

1986年 2月　河西 朝雄

# 目 次

## 第1章 8086アーキテクチャ



## 第2章 MS-DOSにおける アセンブリ言語プログラム開発の基礎

# 第1章

# 8086アーキテクチャ

8086

## 本章の目的

MS-DOSのマクロアセンブラMASMを使ってアセンブリ言語プログラムを作成する場合に，その基礎としてCPUである8086の知識が必要であることは言うまでもないことです.

この章では，8086の特徴であるセグメントによる記憶領域の管理やアドレッシング・モードの概念と8086の命令セットの中から特に重要と思われる命令について説明します.

初心者がマクロアセンブラMASMを学ぶときに，はじめから8086についてすべての事柄を学ぶ必要はありません．ここで説明する内容の知識さえあれば，第2章以降のアセンブラに関する説明やある程度のプログラミングは理解できるはずです.

# 1.1／8086の特徴

**8086**は，1978年に米インテル社が8ビットCPU 8080/85の設計思想を基礎にして開発した16ビットCPUです．

その主な特徴を次に示します．

- **データバス**を16ビットに拡張し，ワード（2バイト）単位のオペレーションを行っているために処理速度が高速である．また，バイト単位のオペレーションも行える．

- **メモリ空間**が1Mバイト（00000H～FFFFFH）に拡張されている．1Mバイトのメモリ空間は，最大64Kバイトの論理セグメントに分割して扱われる．このためにアドレスはセグメント値とオフセット値により管理されるが，セグメント・レジスタにセグメント値を設定すればオフセット値のみで管理できる．

- **アドレッシング・モード**が強化されたので，大きく分けて次の4種類のアドレッシング・モードが可能である．

  直接アドレッシング・モード　(ARRAY)
  レジスタ間接アドレッシング・モード　([BX])
  インデックス・アドレッシング・モード　(ARRAY [SI])
  ベース・インデックス・アドレッシング・モード　([BX＋SI])

- **内部レジスタ**は8080/85のレジスタを拡張した構成になっている．

- **乗除算命令／ストリング処理命令**などが強化されている．

- **コ・プロセッサ（協調プロセッサ）**をサポートしている．

# 1.2 ／ 8086のレジスタ群

8086の内部レジスタ群を図1.1に示します．

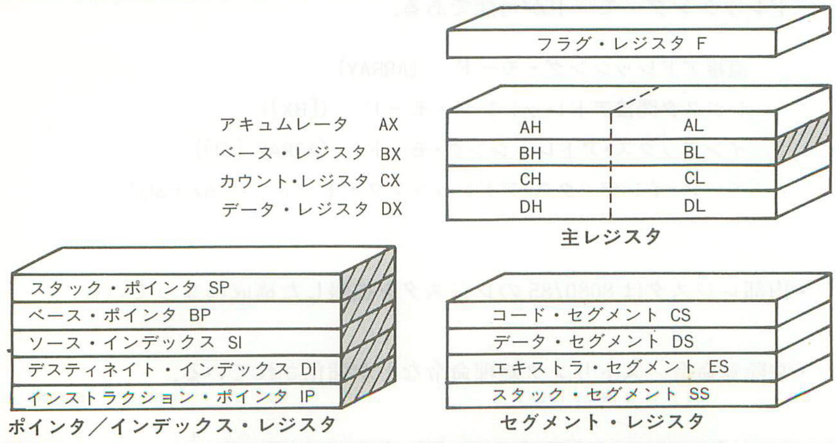
## ■主レジスタ

8086のレジスタは，すべて16ビットで構成されています．その中でA，B，C，Dという4つの主レジスタは，データを16ビット長としても8ビット長としても使用できます．

16ビット長として用いるときには，各レジスタ名の後にXをつけてAX，BX，CX，DXと書きます．8ビット長として用いるときは1つのレジスタを上位8ビット，下位8ビットに分けて，各レジスタ名の後に上位バイト・レジスタならH，下位バイト・レジスタならLをつけて，AH，AL，BH，BL，CH，CL，DH，DLと書きます．

各レジスタは多くの点で対等ですが，AXは主にアキュムレータ用，BXはアドレス・ポインタ用，CXはカウンタ用，DXはデータ用というように使い分けています．フラグ・レジスタは16ビット長のものが1個用意されています．

図1.1　8086レジスタ群



**■ポインタ／インデックス・レジスタ**

8086にはCPUへの命令をチェックするために用いるインストラクション・ポインタIP，スタック（Stack：棚）のトップ・アドレスを指すスタック・ポインタSP，メモリ中のデータを指すデータ・ポインタSI，DI，BPがあります．なお，データ・ポインタとしては主レジスタのBXを利用できます．**図1.1**ではポインタとして利用できるレジスタを斜線で表してあります．

**■セグメント・レジスタ**

1 MBのメモリ空間をセグメントに分割して管理するために必要なレジスタでCS，DS，ES，SSの4つがあります．

## 1.3／8086のセグメンテーション

8080/85，Z80のような8ビットCPUでは直接アクセスできるメモリ空間は64 KBでしたが，8086では1 MBになっています．そのために最大64 KBを1つの論理単位（これを**論理セグメント**と呼ぶ）として扱います．つまり，1 MBのメモリ空間をセグメント単位に64 KBずつ区切って管理しています．

8080/85，Z80では**コード**（MOV，JZなどの命令部分）と**データ**（DB，DWなどで定義される部分）を同一のセグメントに割り当てていましたが，8086ではこれらを別々のセグメントに割り当てることができます．

8086ではプログラムを格納するときの基本論理単位をコード（命令），データ，スタックの3つに分類します．したがって14ページ**図1.2**に示すように，**データ部の拡張（エキストラ）部**と合わせて4つのセグメント・レジスタCS，DS, ES, SSはこれらのセグメントの先頭アドレスを指し示すために用います．

## 1.4／物理アドレスとオフセット

8086では，メモリ管理に**セグメント**という概念を用いています．このためメモリを参照するときには，**物理アドレス**（00000 H～FFFFFH）を用いずにセグメント値とオフセット値という2つの値を用いるという方式を採っています．

図1.2　8086のセグメンテーション

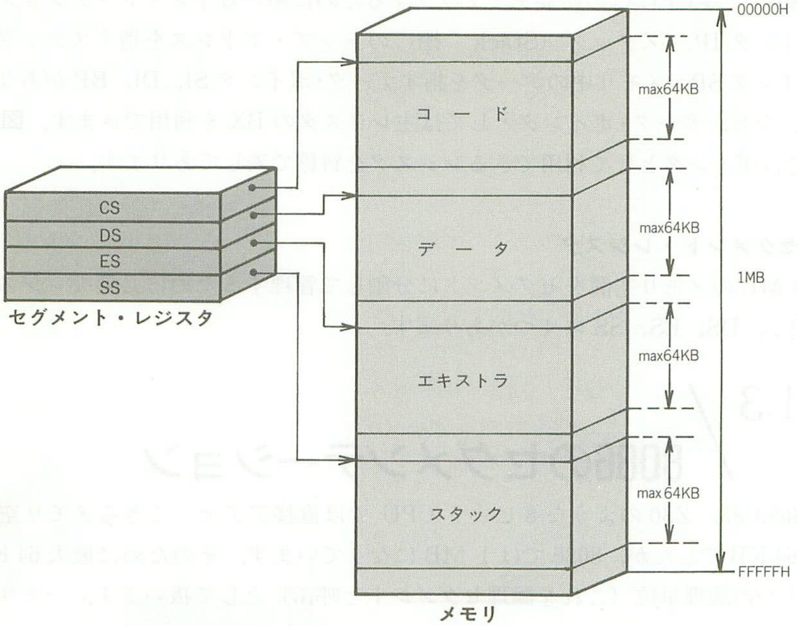



**図1.3**にセグメント値，オフセット値の例を示します．この例ではデータ・セグメントを物理アドレス10000 Hから始めていますが，この値をレジスタにそのまま入れることはできません．その理由は，16ビット長である各レジスタに20ビット長のアドレス値を入れることはできないからです．そこで物理アドレスの下位4ビットを削除して扱うことにします．これを**セグメント値**と呼び，この1000 HをレジスタDSに入れます．セグメント内のアドレスはセグメントの先頭からの距離で表し，これを**オフセット値**と呼びます．

セグメント値は物理アドレスの下位4ビットが0のところから始めなければならないので，その値を物理アドレスで表すと00000 H，00010 H，00020 H，…FFFF 0 Hといった16番地 ($2^4$) 飛びのアドレスとなります．これを**パラグラフ境界**といいます．

オフセット値は0000 H〜FFFFHまでの値を取るので，1つの論理セグメントが扱うことのできる最大メモリは64 KBとなります．

**図 1.3** に示した例では，データ・セグメント内のオフセット 2000 H のアドレスは，

DS：2000 または，1000：2000

と表します．この形式から物理アドレスを求めるには次のような計算を行います．

```
                    ┌──── セグメント値
     1 0 0 0 0 ←──── 0をつける
 +)    2 0 0 0 ←──── オフセット値
 ─────────────────────────
     1 2 0 0 0 ………… 物理アドレス
```

8086 の命令では最初にセグメント・レジスタ（CS，DS，ES，SS）にセグメント値をセットしておけば，各セグメント内のアクセスではオフセット値だけをオペランドとして扱うことができます．

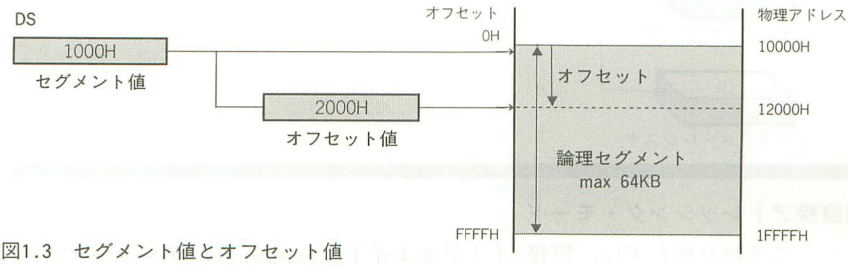


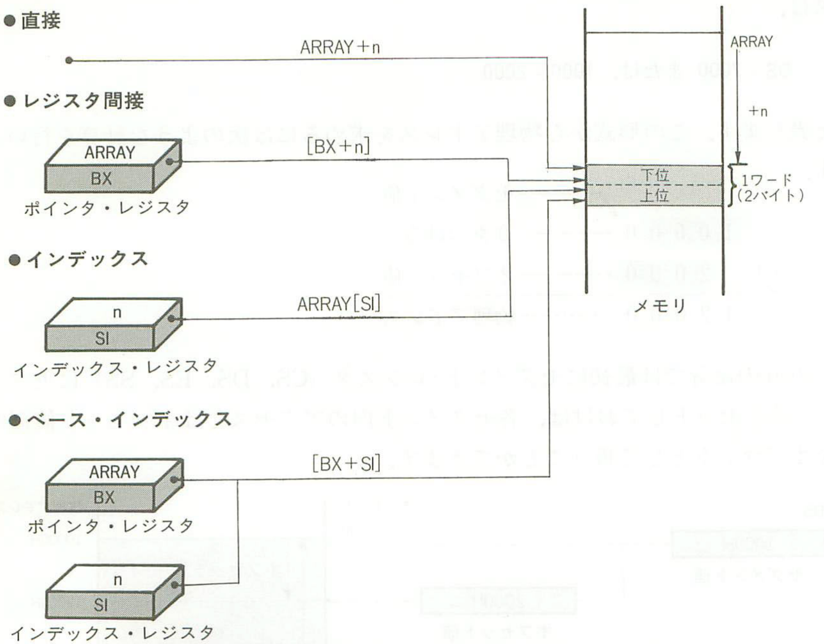

図1.3　セグメント値とオフセット値

# 1.5／アドレッシング・モード

機械語のプログラムは高級言語の場合と異なり，プログラムのあちこちでメモリに格納されているデータを参照することが多くあります．このメモリ上のデータを参照する方法を**アドレッシング・モード**と呼び，8086 では大きく分けて次の 4 種類のアドレッシング・モードがあります．

なお, 各モードの説明を**図 1.4** に示すような ARRAY 番地から n 番地離れた番地の内容を参照する場合を例として行います. ここでは ARRAY にワード属性が与えられ，16 ビット長として扱われているものとします．

図1.4　アドレッシング・モード




## ■直接アドレッシング・モード

レジスタ類を使わずに，**直接(イミディエイト：immediate)データ** ARRAY と n を用いて，次のように表します．

```
ARRAY＋n
```

たとえば次のように命令すると，ARRAY＋n 番地の内容がレジスタ AX にロードされます．

```
MOV     AX,ARRAY＋n
```

もう少し詳しく説明すると，直接メモリにデータを格納するときは，ARRAY がワード属性(16 ビット長)になっているので，ARRAY＋n 番地の 1 バイトデータがレジスタ AL に，ARRAY＋n＋1 番地の 1 バイトデータがレジスタ AH にロードされます．

また，ARRAYはメモリに格納された内容を指し示し，ARRAY番地というアドレス値（オフセット値）を指し示しているわけではない点に注意してください．オフセット値を表すには，次のように**OFFSET 演算子**を用います．

```
MOV    AX,OFFSET  ARRAY
```

**■レジスタ間接アドレッシング・モード**

ARRAYという値（アドレス値）を**ポインタ・レジスタ**（たとえばBX）にセットしておき，

[BX＋n]

と表します．

```
MOV    BX,OFFSET  ARRAY
MOV    AX,[BX＋n]
```

と命令すると，レジスタBXの値に＋nした番地の内容がレジスタAXにロードされます．

この方法の利点はプログラムの中でBXの内容を変えていくことができるので，メモリを順にアクセスしていく必要があるときなど，INC BXとするだけでよく，プログラムが簡単に記述できます．一方，先のARRAY＋nを用いる方法は命令の意味が単純で分かりやすい反面，このようなプログラムへの応用が効きません．

ポインタ・レジスタにはBXとBPがよく使われます．ベース・レジスタBXをアドレッシング・モードに使用すると，データ・セグメントDSがディフォルト用のセグメント・レジスタとして使われ，ベース・ポインタBPをアドレッシング・モードに使用すると，スタック・セグメントSSがディフォルト用のセグメント・レジスタとして使われます．

**■インデックス・アドレッシング・モード**

変位量n（ARRAY先頭からの距離）を**インデックス・レジスタ**，たとえばSIにセットしておき，

```
ARRAY [SI]
```

とすると，高級言語の1次元配列に似た記述ができます．これを**インデックス・アドレッシング**と呼びます．次のように命令すると，ARRAYのn番目の要素がレジスタAXにロードされます．

```
MOV    SI,n
MOV    AX,ARRAY [SI]
```

インデックス・レジスタとしてはSI（ソース・インデックス），DI（デスティネート・インデックス）が使用できます．

**■ベース・インデックス・アドレッシング・モード**

ARRAYとnの両方を，各々ポインタ・レジスタおよびインデックス・レジスタにセットしておき，

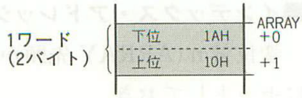
```
[BX+SI]
```

としてレコード（構造体）などのデータの構造を表すために用います．これを**ベース・インデックス・アドレッシング**と呼びます．次のように命令すると，ARRAYのn番目の要素がレジスタAXにロードされます．

```
MOV    BX,OFFSET ARRAY
MOV    SI,n
MOV    AX,[BX+SI]
```

## 1.6 / バイト・オペレーションとワード・オペレーション

ARRAY番地がARRAY　DW　101AHのように**ワード属性**（データをワード単位で扱う指定）で定義されているとき，




```
MOV    AX,ARRAY
```

という命令は**ワード・オペレーション**となり，ARRAY番地を下位バイト，ARRAY＋1番地を上位バイトとした1ワード・データをレジスタAXにロードします．したがってレジスタALに1AH，レジスタAHに10Hがロードされたということになります．

またARRAYをワード属性で定義しているときに，

```
MOV    AL,ARRAY
```

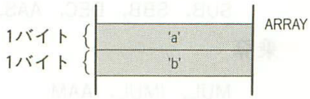
と命令するのは誤りです．この命令のALの属性が8ビット長のバイト型であるため，16ビット長のワード型であるARRAYの内容（2バイト）を格納することができないからです．

このような場合は，**PTR（ポインタ演算子）**を用いて，

```
MOV    AL,BYTE   PTR   ARRAY
```

と書いて一時的に（この命令が実行されるときだけ）ARRAYの属性をバイト型に変えます．

次にARRAY番地がARRAY　DB　'ab'のように**バイト属性**で定義されているとき，




```
MOV    AL,ARRAY
```

という命令は**バイト・オペレーション**となり，レジスタALにARRAY番地の1バイト・データ'a'がロードされます．ワード・オペレーションで行いたければ，ポインタ演算子PTRを用いて次のように命令します．

```
MOV    AX,WORD   PTR   ARRAY
```

# 1.7／8086命令セット

16ビットCPU 8086の命令セットを次に示します．これらの命令についてすべてを説明することは本書の目的ではないので，本書を読み進めていく上で必要なものを次節から説明していきます．

■転送

汎用転送／交換

MOV，XCHG

アドレス・オブジェクト転送

LEA，LDS，LES

スタック転送

PUSH，POP，PUSHF，POPF

入出力

IN，OUT

その他

LAHF，SAHF，XLAT

■算術

加算

ADD，ADC，INC，AAA，DAA

減算

SUB，SBB，DEC，AAS，DAS，NEG，CMP

乗算

MUL，IMUL，AAM

除算

DIV，IDIV，AAD，CBW，CWD

■論理

シングル・オペランド

NOT，SHL，SHR，SAL，SAR，ROL，ROR，RCL，RCR

ダブル・オペランド

AND，TEST，OR，XOR

■ストリング

リピート・プリフィックス

REP，REPE，REPZ，REPNE，REPNZ

プリミティブ・ストリング

MOVS，MOVSB，MOVSW，CMPS，SCAS，LODS，STOS

■分岐

無条件分岐

JMP，CALL，RET

条件分岐

JA/JNBE，JAE/JNB，JB/JNAE，JBE/JNA，JC，JE/JZ，JG/JNLE，
JGE/JNL，JL/JNGE，JLE/JNG，JNC，JNE/JNZ，JNO，JNP/JPO，
JNS，JO，JP/JPE，JS

繰り返し

LOOP，LOOPZ，LOOPE，LOOPNZ，LOOPNE，JCXZ

割り込み

INT，INTO，IRET

■CPU 制御

フラグ制御

CLC，CMC，STC，CLD，STD，CLI，STI

CPU 制御

HLT，WAIT，ESC，LOCK，NOP

なお，8086 命令セットを巻末に付録として掲載したので参考にしてください．

# 1.8 / 転送命令(MOV)

レジスタとレジスタ間，メモリとレジスタ間などで，データを転送するのがMOV（move）命令です．8086の命令の中では，もっとも頻繁に利用される基本的な命令です．この命令の一般形は，

```
MOV    destination,source
```

で，データはsource（源泉）→ destination（目的地）へ移動します．なお，転送といっても，転送後もsource（源泉）にあるデータはそのまま残り，なくなることはありません．




転送元と転送先であるdestinationとsourceのオペランドの組み合わせには，次の9通りがあります．

reg, reg

reg, mem

reg, imm

mem, reg

mem, imm

sreg, reg

sreg, mem

reg, sreg

mem, sreg

注）reg…汎用レジスタ：AL, AH, BL, BH, CL, CH, DL, DH,
AX, BX, CX, DX, SP, BP, SI, DI
mem…メモリ：WADR, BADR, [BX], BADR [SI] など
imm…16/8ビット長の即値
sreg…セグメント・レジスタ：CS, DS, ES, SS

MOV命令を使う上で，誤りやすい点は次のような事項です．

・destination側とsource側のオペランドのデータ長(バイト型：8ビット，ワード型：16ビット）の区別が異なっていてはならない
　たとえばWADRが,

```
WADR   DW 1000 H
```

のようにワード属性で定義されているときに,

```
MOV   AL,WADR
```

とするのは誤りです．この場合は，PTR演算子を用いて,

```
MOV   AL,BYTE   PTR   WADR
```

とします．
　また，次のような命令,

```
MOV    [BX],0000 H
```

は［BX］で示されるアドレスに0000 Hをワード転送しますが,

```
MOV    [BX],0 H
```

では，0 Hがワード型かバイト型かはっきりしないので，MASM（マクロアセンブラ）はこれを認めません．
　もし，バイト転送したいのであれば,

```
MOV   BYTE   PTR [BX],0 H
```

とします．

　このように[BX]などがオペランドに用いられる場合，それだけではワード型なのかバイト型なのかはっきりしないので,アセンブラはsource側のオペランドの型に合わせます．もし，相手のオペランドが0 Hのようにワード型かバイト型かはっきりしない場合は,PTR演算子などで型を明示しなければなりま

せん.

・BADRやWADRという名前をオペランドに用いた場合は，番地のアドレス値（オフセット値）ではなく，その番地に記憶されているメモリの内容を表す

たとえば次のような命令，

```
MOV    AX,WADR
```

はWADR番地の内容をレジスタAXに転送するのに対し，

```
MOV    AX,OFFSET WADR
```

では，WADR番地というアドレス値(オフセット値)を転送します．同じ操作をLEA命令を用いて次のようにも書くことができます．

```
LEA    AX,WADR
```

LEA命令は2番目めのオペランドのオフセット値を1番目のオペランドに転送します．

・セグメント・レジスタに即値データを転送することはできない

たとえば次のように，

```
MOV    DS,OFFSET WADR
```

という転送はできないので，汎用レジスタを中継します．

```
MOV    AX,OFFSET WADR
MOV    DS,AX
```

・レジスタ間接アドレシングには［AX］，［CX］，［DX］などは使えない

たとえば，次のような記述はできません．

```
MOV    DS,[AX]
```

**【転送命令の例】**

```
MOV   AX,CX       ………… AX ← CX
MOV   BL,CH       ………… BL ← CH
MOV   DX,WADR     ………… DX ← WADR 番地を先頭とする 1 ワードデータ
```




```
MOV   CX,[BX]      ………… CX ← BX 番地を先頭とする 1 ワードデータ
MOV   BADR,AL      ………… BADR 番地← AL
MOV   BX,1000 H    …………BX ← 1000 H
```

# 1.9／加減算命令(ADD／SUB)

2 つのオペランドの内容を加算する ADD ／ ADC 命令，2 つのオペランドの内容を減算する SUB ／ SBB 命令について説明します．これらの命令のオペランドの組み合わせは次の 5 通りです．

reg, reg
reg, mem
reg, imm
mem, reg
mem, imm

## ● ADD ／ ADC

**ADD（Addition）命令**は，第 1 オペランドと第 2 オペランドの内容を加算し，その結果を第 1 オペランドに格納します．

**ADC（Addition with carry）命令**は，キャリーフラグの内容も含めて加算する命令で，4 バイト，8 バイトなどの桁数の多い加算を行う場合に用います．

**【加算命令の例】**

```
ADD   BX,DX         ………… BX ← BX+DX
ADD   AL,AH         ………… AL ← AL+AH
ADD   DX,0005 H     ………… DX ← DX+0005 H
ADD   WADR,CX       ………… WADR 番地の内容に CX の内容を加える
```

### ● SUB ／ SBB

**SUB (Subtract) 命令**は，第1オペランドから第2オペランドの内容を減じ，その結果を第1オペランドに格納します．

**SBB (Subtract with borrow) 命令**は，キャリーフラグの内容も含めて減算する命令で，4バイト，8バイトなどの桁数の多い減算を行う場合に用います．

**【減算命令の例】**

```
SUB   BX,DX         ………… BX ← BX−DX
SUB   AL,AH         ………… AL ← AL−AH
SUB   DX,0005 H     ………… DX ← DX−0005 H
SUB   WADR,CX       ………… WADR 番地の内容から CX の内容を引く
```

# 1.10 / インクリメント/ディクリメント命令(INC/DEC)

プログラム上では，レジスタやメモリの内容を＋1（Increment）したり，－1（Decrement）したりすることがよくあります．そのため，

```
ADD   AX,0001
SUB   BX,0001
```

などの加減算命令を用いる代わりに，それ専用の命令として，**インクリメント／ディクリメント命令**が用意されています．

書式は次のようになります．

```
INC/DEC   オペランド
```

オペランドの内容を＋1，－1して，その結果を格納します．オペランドにはreg，memを指定することができます．

**【INC/DEC命令の例】**

```
MOV   AX,0001
DEC   AX…………………………   AXの内容は－1されて，0になる．このとき，Zフラ
                             グがセットされる
INC   BL…………………………… BLの内容が＋1される
INC   BYTE  PTR [BX] …… BXで示されるメモリ（バイト属性）の内容が＋1される
DEC   WORD  PTR [BX]…… BXで示されるメモリ（ワード属性）の内容が－1される
```

# 1.11 乗除算命令(MUL／DIV)

8080/Z80にはなかった乗算命令と除算命令が，8086では使えるようになりました．

## ● MUL（Multiply accumulator by register/memory）

乗算命令の書式は次のように，

```
MUL　オペランド
```

と書き，オペランドがバイト長ならばレジスタALとオペランドの内容が乗算され，その積がレジスタAXに格納されます．

AX ⇦ AL × オペランド（バイトオペレーション）

オペランドがワード長ならば，レジスタAXとオペランドの内容が乗算され，その積がレジスタDX，AXに格納されます（ワードオペレーション）．

DX AX ⇦ AX × オペランド
（ワードオペレーション）

オペランドとしては，

reg …… 16/8ビット汎用レジスタ
mem …… メモリ

が指定できます．なお，次の点には注意してください．

- オペランドには即値データは使用できない
- 被乗算レジスタ(オペランドとしては特に書かない)はALまたはAXに限定されている

【MUL 命令の例】

```
        MOV   AL, 5
        MUL   COUNT
          :
COUNT  DB   2
        MUL   CX
```

(MOV AL, 5 〜 COUNT DB 2) AL×COUNT，つまり5×2の積がレジスタAXに格納される

MUL CX ← DX：AX ← AX × CX

● IMUL (Integer Multiply)

この命令は，**符号付き数の乗算**を行う他は，MUL 命令に準じます．

● DIV (Division)

**除算命令**の書式は次のように，

DIV　オペランド

と書き，オペランドがバイト長ならば，レジスタ AX÷オペランドの除算が行われ，商が AL，余りが AH に格納されます．

| | | | | | |
|---|---|---|---|---|---|
| 商 | AL | ⇦ | AX | ÷ | オペランド |
| 余り | AH | | | | |

オペランドがワード長ならば，レジスタ DX(上位バイト)，AX(下位バイト)をオペランドで割った商が AX に，余りが DX に格納されます．

| | | | | | | |
|---|---|---|---|---|---|---|
| 商 | AX | ⇦ | DX | AX | ÷ | オペランド |
| 余り | DX | | | | | |

オペランドとしては，

reg …… 16/8 ビット汎用レジスタ

mem …… メモリ

が指定できます．なお，次の点には注意してください．

・オペランドには即値データは使用できない
・被除算レジスタ(オペランドとしては特に書かない)はAXまたはDX：AXに限定されている

また，除算の結果がレジスタに入り切らない場合にはタイプ0の割り込みが発生します．

【DIV 命令の例】

```
        MOV  AL,1000 H
        DIV  COUNT        AL÷COUNT，つまり1000H÷2の商がレジスタAL，
                          余りがAHに格納される
             ⋮
COUNT   DB   2

        DIV  CX ←——— DX：AX÷CXを行い，商がAXに，余り
                     がDXに格納される
```

● IDIV（Integer Division）

この命令は，**符号付き数の除算**を行う他は，DIV 命令に準じます．

# 1.12／論理演算命令

8086にはビット毎の**論理演算**を行う命令として，AND，OR，XOR，TEST，NOTが用意されています．

AND，OR，XOR，TESTは**ダブル・オペランド**で，次の5通りのオペランドの組み合わせがあります．

reg，reg
reg，mem
reg，imm
mem，reg
mem，imm

NOTは**シングル・オペランド**で，

reg
mem

が指定できます．

AND，OR，XORの真理値表を次に示します．

AND

| a | b | c |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

OR

| a | b | c |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

XOR

| a | b | c |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

### ● AND

2つのオペランドのビット毎のANDを取り，その結果を第1オペランドに格納します．

```
MOV   BL,55 H         マスク
                      される──→01010101    (55 H)
AND   BL,0FH          AND  ) 00001111    (0 FH)
                           ─────────────────────
                             00000101    (05 H)
```

ANDを取る相手が'0'の場合は，何であっても結果が0になるので，ANDは特定ビットを'0'にマスクするときによく使われます．たとえば第7ビットだけをマスクしたければ，7FHとのANDを取ればよいことになります．

### ● OR

2つのオペランドのビット毎のORを取り，その結果を第1オペランドに格納します．

```
MOV   BL,55 H               01010101    (55 H)
OR    BL,0FH          OR  ) 00001111    (0 FH)
                          ─────────────────────
                            01011111    (5 FH)
```

ORを取る相手が'1'の場合は，何であっても結果が1になるので，ORは特定ビットを'1'にするときによく使われます．

### ● XOR（Exclusive OR）

2つのオペランドのビット毎のXORを取り，その結果を第1オペランドに格納します．

```
MOV   BL,8 FH               10001111    (8 FH)
XOR   BL,0 FFH        XOR ) 11111111    (FFH)
                          ─────────────────────
                            01110000    (70 H)
```

'1'とのXORを取る場合は，相手のビットを反転させるので，XORは特定ビットのビット反転を行うときによく使われます．

また，

```
XOR   AL,AL
```

は，ALの値が何であっても，同じもの同士のXORを取れば結果は0になるので，XORはオペランドをゼロクリアする場合にも使われます．

## ● TEST

ANDと同じ動作をしますが，結果を第1オペランドに格納しない点が異なります．ビットテストによる条件分岐に使われます．

```
TEST   BL,4
                     7xxxxx□xx0   BL
            AND)      00000100    (4)
                 ─────────────────────
                      00000□00
```

レジスタBLの第2ビットが'0'ならば，ANDの結果は0になり，Zフラグがセットされます．'1'ならばZフラグはセットされません．これを利用して，条件分岐命令JZ／JEなどによって分岐を行います．

## ● NOT

オペランドのビットをすべて反転します．これは別の言い方をするとオペランドの1の補数を作るということになります．また，2の補数（ビット反転＋1）をつくるにはNEG命令を用います．

```
MOV   BL,01 H          00000001    (01 H)
NOT   BL                  ⇩ NOT
                       11111110    (FEH)
```

# 1.13 ／ ローテイト／シフト命令

8ビット／16ビットのデータを，最初のビットと最後のビットが連続したひとつの輪のように考え，第2オペランドに示される数だけ左，または右に回転させる操作を**ローテイト命令**と呼びます．

また，8ビット／16ビットのデータをひとつの線のように考え，第2オペランドに示される数だけ左，または右に移動させる操作を**シフト命令**と呼びます．以下，ローテイト／シフト命令について説明します．

たとえば，SHLは左へシフトする命令であり，

```
SHL　第1オペランド，第2オペランド
```

と書き，第1オペランドのデータを第2オペランドで示される数だけ左へシフトします．

第1オペランドとしては，reg，memが指定できますが，第2オペランドは1かCLだけしか使用できません．したがって，1ビットのローテイト／シフト以外はシフトしたい数をCLレジスタに格納してから行うことにします．

### ● SHL／SAL（Shift logical left／Shift arithmetic left）




ビット列を左へ移動させます．左へシフトした結果，空白になった右端には0が入り，押し出された形の左端のビットはキャリー・フラグCFに入ります．SHLとSALは同じ機能です．

### ● SHR（Shift logical right）



ビット列を右へ移動させます．右へシフトした結果，空白になった左端には0が入り，押し出された形の右端のビットはキャリー・フラグCFに入ります．この命令は符号なし数を1/2にする場合に用いると便利です．

● SAR（Shift arithmetic right）



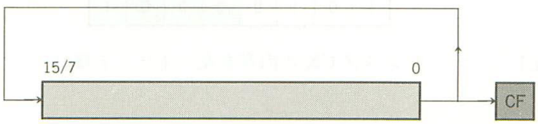

ビット列を右へ移動させます．右へシフトしても最上位ビット（サインビット）はそのまま残ります．押し出された形の右端のビットはキャリー・フラグCFに入ります．この命令は符号付き数を1/2にする場合に用いると便利です．

● ROL（Rotate left）



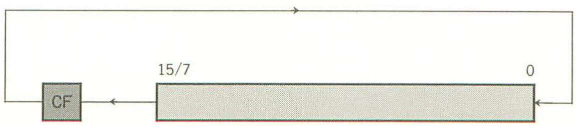

ビット列を左へ回転させます．左へ回転した結果，最上位ビットは最下位ビットに回り，さらにキャリー・フラグCFにも入ります．

● ROR（Rotate right）



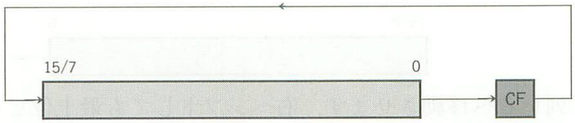

ビット列を右へ回転させます．右へ回転した結果，最下位ビットは最上位ビットに回り，さらにキャリー・フラグCFにも入ります．

● RCL（Rotate through carry left）



キャリー・フラグを含めて，ビット列を左へ回転させます．左へ回転した結果，最上位ビットはキャリー・フラグCFに，キャリー・フラグの値は最下位ビットに入ります．

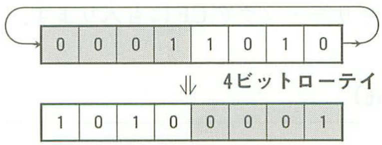
● RCR（Rotate through carry right）




キャリーフラグを含めて，ビット列を右へ回転させます．右へ回転した結果，最下位ビットはキャリー・フラグCFに，キャリー・フラグの値は最上位ビットに入ります．

**【ローテイト／シフト命令の例】**

```
MOV   BL,1AH
MOV   CL,4
ROR   BL,CL   …… BLを4ビット右へ回転（ローテイト）する
```




```
SHL   CX,1    …… レジスタCXの内容を左へ1ビット移動（シフト）する．
```

# 1.14／比較命令(CMP)とフラグの変化

比較命令 CMP（Compare）の動作そのものは減算命令の範疇に含まれますが，普通，比較命令はその減算結果によりフラグをセットし，その直後の条件分岐命令に用いられる条件を設定することに使用されます．

たとえば次のような例を考えてみることにします．

```
        CMP   AX,BX
        JE    ABC
              ⋮
ABC：
```

CMP AX,BX では，AX−BX の減算が行われ，その結果によりフラグがセットされます．もし，AX＝BX ならばZ（ゼロ）フラグがセットされます．

JE　ABC 命令はZフラグをチェックし，もしセットされていたならば，指定された番地 ABC へジャンプします．

これらの2つの命令を合わせて，「もし，AX＝BX ならば ABC へ分岐する」という意味を持つと考えればよいことになります．

条件を判定するフラグは，条件 cc より異なりますが，条件 cc の種類については次節 1.15 を参照してください．

CMP 命令のオペランドは加減算命令の場合と同じように，次の5種類が指定できます．

reg，reg

reg，mem

reg，imm

mem, reg

mem, imm

CMP命令の動作は第1オペランドから第2オペランドを減じ，その結果でフラグをセットします．減算結果を第1オペランドに格納しない点がSUB命令とは異なります．

CMP命令以外にも，加減算，乗除算，論理演算などの演算結果によってもフラグは変化するので，それらの命令によりフラグをセットした後で条件分岐を行う場合も多くあります．

| | 15 | | | | | | | | | | | | | | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| フラグ・レジスタF | | | | | O | D | I | T | S | Z | | A | | P | | C |

フラグの変化は，各命令により多少異なるので，どのように変わるか詳しくは巻末の付録を参照してください．なお，一般的なフラグの変化を次に示します．

● **CPUの状態を表すステータス・フラグ（状態フラグ）**

CF：キャリー・フラグ……最上位ビットでキャリーが発生したか否かを判定

PF：パリティ・フラグ……演算結果の'1'のビット数が偶数か奇数か判定

AF：ハーフ・キャリー・フラグ……下位4ビットでキャリーが発生したか否か判定

ZF：ゼロ・フラグ……演算結果が0か否かを判定

SF：サイン・フラグ……演算結果が正か負かを判定

OF：オーバーフロー・フラグ……演算結果がオーバーフローか否かを判定

● **CPUの動作を制御するコントロール・フラグ（制御フラグ）**

DF：ディレクション・フラグ……ストリング命令の際に移動方向を指定

IF：インタラプト・イネーブル・フラグ……外部割り込みが可能か否かを判定

TF：トラップ・フラグ……デバッグのためのシングルステップ動作を指定

それでは，ここで今まで説明した命令についてフラグの変化を参考までにま

とめておきます.

| | O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|---|
| MOV | | | | | | | | | |
| ADD ／ ADC ／ SUB ／ SBB | × | | | | × | × | × | × | × |
| INC ／ DEC | × | | | | × | × | × | × | |
| MUL | × | | | | U | U | U | U | × |
| AND/OR/XOR/TEST | 0 | | | | × | × | U | × | 0 |
| NOT | | | | | | | | | |
| CMP | × | | | | × | × | × | × | × |

注）空白：変化を受けない, ×：変化を受ける, U：未定, 0：0にセットする.

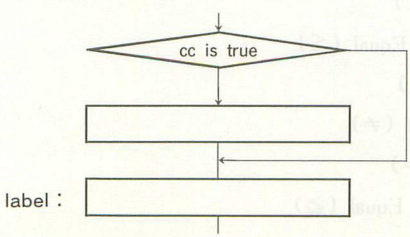
# 1.15 ／ 条件分岐命令(Jcc)

プログラムの流れを条件により変更できるのが, **条件分岐命令**です. 条件に応じていくつかの分岐命令があります. 一般的には,

```
Jcc   label
```

と書き, 条件ccを満たしたときにlabelの番地にジャンプして分岐します. 条件が満たされなければ, そのままJcc命令の次の命令に進みます.

条件分岐命令をフローチャートで示すと次のようになります.

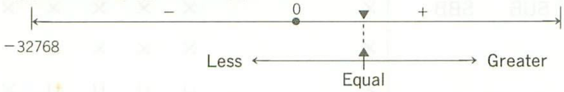



条件分岐命令で指定できるlabelは, ショート・ラベルだけです. したがって, 分岐するときの飛び先は, ±128バイトの範囲に限られます.

分岐命令に用いる条件ccの書き方はJcc命令の前に行う比較命令などの演

算結果を，符号付きの数として扱っているか，符号なしの数として扱っているかで次のように分類できます．

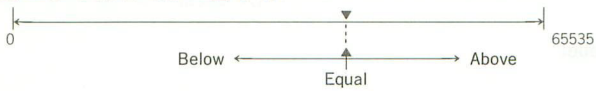
**●符号付きの数による条件ccの書き方**




大小関係の条件は大きい（G：Greater），等しい（E：Equal），小さい（L：Less）を示すG，E，Lを用いて次のように書きます．

L　…　Less（＜）
LE　…　Less or Equal（≦）
E　…　Equal（＝）
NE　…　Not Equal（≠）
G　…　Greater（＞）
GE　…　Greater or Equal（≧）

**●符号なしの数による条件ccの書き方**




大小関係の条件は大きい（A：Above），等しい（E：Equal），小さい（B：Below）を示すA，E，Bを用いて次のように書きます．

B　…　Below（＜）
BE　…　Below or Equal（≦）
E　…　Equal（＝）
NE　…　Not Equal（≠）
A　…　Above（＞）
AE　…　Above or Equal（≧）

大小関係の比較だけならば，上に挙げた条件ccの書き方だけで十分です．

また，分岐命令に用いる条件ccの書き方には，これ以外にもC（キャリーが生じた場合），Z（ゼロになった場合），NC（キャリーが生じない場合），NZ（ゼ

ロでない場合）などのフラグの名称を用いた直接の意味を示すものもあり，それを以下に示します．

C　…　キャリーが生じた（減算の結果が負になったりオーバーフローになった）場合
NC　…　キャリーが生じない場合
Z　…　ゼロになった場合
NZ　…　ゼロでない場合
O　…　オーバーフローした場合
NO　…　オーバーフローしていない場合
S　…　サインフラグが１になった場合
NS　…　サインフラグが０になった場合
P　…　パリティフラグが１になった場合
NP　…　パリティフラグが０になった場合

**【条件分岐命令の例】**

```
        CMP    BL,5
        JB     ABC    …… BL＜5ならばABCに分岐する
                 :
ABC：

        TEST   CL,80H
        JE     ABC    …… CLの第7ビットが0ならばABCへ分岐する
                 :
ABC：
```

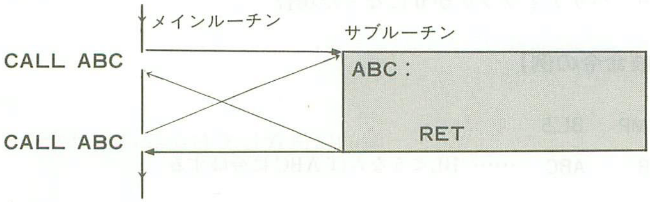
この命令はJEの代わりにJZと書いてもよく，JZ（ゼロならば）の方が命令の意味がわかりやすい．

# 1.16 / サブルーチン命令(CALL/RET)

何度も使う処理やある一定の決まった手順の処理は，ひとまとめにしておき，必要になったときに呼び出して用いるようにしておけば，同じ処理をひとつのプログラムの中で何個所に書かなくてもよくなります．

プログラムの根幹となる部分を**メインルーチン**と呼び，その中で呼び出される副プログラムを**サブルーチン**と呼んでいます．

メインルーチンからサブルーチンを呼ぶときは**CALL 命令**を使い，サブルーチンからメインルーチンに戻るときには**RET 命令**を用います．ひとつのサブルーチンから別のサブルーチンを呼ぶこともできます．




CALL 命令の一般形は次のようになります．

```
CALL   label
```

CALL 命令が実行されると，labelで示される番地から始まるサブルーチンへ分岐します．この飛び先のラベル（label）には**NEAR（ひとつのセグメント内），FAR（セグメント外）**という属性があります．ラベルの属性が不明確な場合は，アセンブラが文脈から判断しますが，その場合，多くのアセンブラでは属性をNEARとして扱います．

またMASMなどのアセンブラは，セグメントがいくつも存在する場合に，サブルーチンを**PROC～ENDP（プロシージャ定義擬似命令）**を用いて定義し，ラベルの属性を明確に指定しておけば誤りが少なくなります．これについては第4章**4.11**を参考にしてください．

なお，CALL 命令ではラベルを指定する直接CALL 以外にも，レジスタ・ポイ

ンタ，メモリ・ポインタを用いたCALLがありますが，ここでは説明を省きます．

**【サブルーチン命令の例】**

```
        CALL  ABC・・・ラベルABCは同一セグメント内にあるのでnear属性
                       として判断される
          ⋮
ABC：  サブルーチン本体
        RET
```

# 1.17／スタック命令(PUSH／POP)

CPUが持っているレジスタの数は限られているため，プログラム実行中に保存しておきたいレジスタの値がレジスタの数を超えてしまうような場合は，外部メモリにそのレジスタの値を一時的に保管しておかなければなりません．このときに保管場所である外部メモリの番地をいちいち管理していたのでは大変なので，メモリ上のある領域に**スタック**（棚）として保管します．

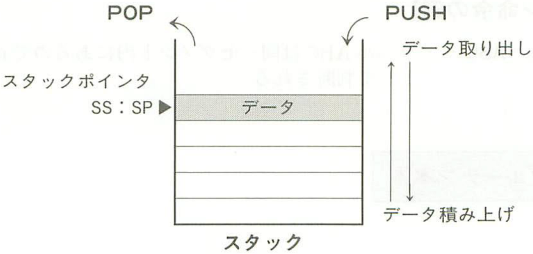
スタックに保管されるデータはスタックの下（上位番地）から上（下位番地）へ向かって順に積まれていき，あるデータが必要になったときには上から順に取り出していきます．この方法でデータを保管しておけば，プログラマなどがデータを格納したのはメモリの何番地であるのかということを気づかう必要がなくなるわけです．ただ，データを積んでいった順序とデータを取り出す順序が逆になるということには注意しなければなりません．

スタックに現在どのくらいデータが格納されているかは，その時点のスタック・ポインタSPを調べればわかります．スタック・ポインタSPは一番最後にデータを格納した番地を示します．

スタックはプログラマがスタック命令で用いる他にも，ある種の命令ではCPUが自動的にそのスタックを利用している場合があります．

たとえばCALL命令が実行されると，分岐先へ飛んだ後，また戻ってくる番

地(戻り番地)がスタックに**退避(PUSH)**されます．またRET命令が実行されると，このスタックに積まれた戻り番地が取り出されます(POP)．スタックはスタック・セグメント・レジスタSSで示されるセグメント上に確保されます．



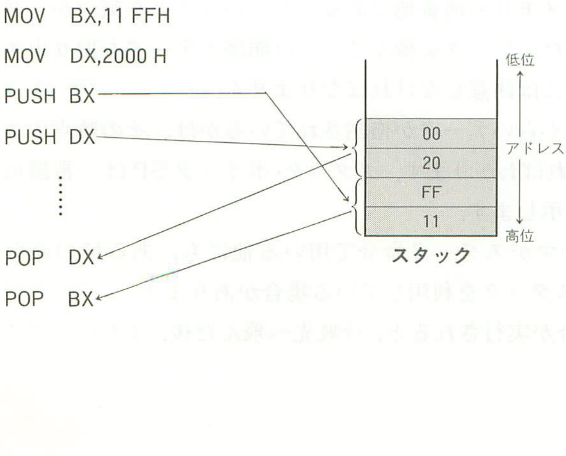

PUSH／POP命令には，次のような3種類のオペランドが使用できます．

```
PUSH/POP   reg
           sreg
           mem
```

ただし，CS（コード・セグメント）レジスタをPOPすることはできません．その理由はプログラマの不注意によりCSレジスタをPOPして，プログラムの暴走が起きてしまうことを防ぐためです．

なお，フラグ・レジスタ関係のスタック命令には，専用の命令PUSHF／POPFを用います．

**【スタック命令の例】**



・BX，DX の順に PUSH したデータを DX，BX の順で POP すれば，元通りのデータが復元できる．

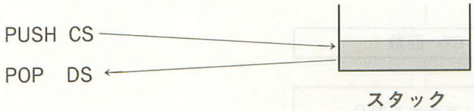
PUSH ／ POP 命令の一般の使い方に反して，PUSH と POP の対応をわざと逆にすることで，スタックを介して CS レジスタの値を DS に転送することができます．MOV 命令ではセグメント・レジスタ間の直接転送を認めていないので，次のような転送はできません．

```
MOV     DS,CS
```

したがって，次のように AX レジスタを介して転送しなければなりません．

```
MOV     AX,CS
MOV     DS,AX
```

そこで PUSH ／ POP 命令を用いて，




とする場合がよくあります．MOV 命令を用いた CS → DS への転送は 4 バイトの機械語コードに落されるのに対して，PUSH／POP 命令を用いた CS → DS への転送は 2 バイトの機械語コードで済みます．

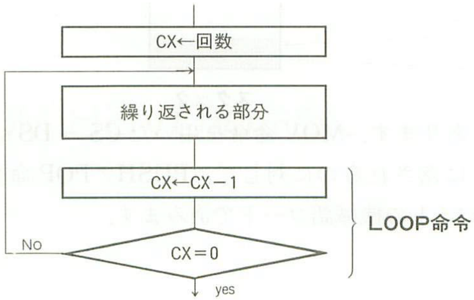
# 1.18 / 繰り返し命令(LOOP)

LOOP命令は，ループの外で初期設定されたレジスタCXの内容を，ループ1回毎に−1し，CXが0になるまでこれを繰り返します．したがって，CXで与えられた回数だけループを回ることになります．

LOOP命令の書式は，

```
LOOP   label
```

と書き，**ラベル(label)**には**ショートラベル(±128バイトの範囲)**だけが使用できます．




**【繰り返し命令の例】**

```
        MOV    CX,100      …… 繰り返しルーチンAを100回繰り返す
ABC：
       [繰り返しルーチンA]
       LOOP   ABC
```

また，次のように2重ループを作ることもできます

```
       MOV    CX,100
```

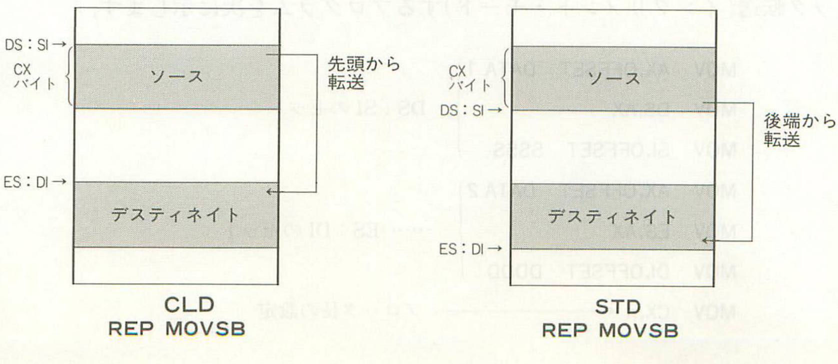
```
XYZ：  PUSH  CX          …… 繰り返しルーチンAを100回，繰り返し
                             ルーチンBを100×10回繰り返す
       ┌──────────────────┐
       │ 繰り返しルーチンA │
       └──────────────────┘
       MOV  CX, 10
ABC：  ┌──────────────────┐
       │ 繰り返しルーチンB │
       └──────────────────┘
       LOOP  ABC
       POP   CX
       LOOP  XYZ
```

・PUSH CX，POP CX 命令は，外側のループの CX を保持するために用いている．

# 1.19／ストリング命令(MOVSB/MOVSW)

8086 ではデータのブロック転送を行うための**ストリング命令**が用意されています．ここではそれらの命令のうち，代表的な MOVSB ／ MOVSW 命令とそれを補助する STD ／ CLD 命令について説明します．

次のようなデータのブロック転送を行う場合を考えてみることにします．



**MOVSB（Move string byte）命令**はDS：SIで示されるソース・データブロックのデータをES：DIで示されるディスティネイト・データブロックにバイト単位で転送します．

MOVSB命令が実行されるたびに，SI，DIはインクリメント，またはディクリメントされます．インクリメント・モードならばブロック先頭から，ディクリメント・モードならばブロック後端からブロック転送を行います．

MOVSB命令を実行する際に，このインクリメント・モード，ディクリメント・モードを設定するのがSTD／CLD命令で，MOVSB命令の実行に先立ち，STDかCLD命令のいずれかを実行し，モードを設定します．

CLD ……　SI/DIインクリメント・モード（ブロック先頭から転送）
STD ……　SI/DIディクリメント・モード（ブロック後端から転送）

普通，MOVSB命令などのストリング命令は，繰り返し動作と共に使用されます．これを行うのが**リピート・プリフィックス命令（REP）**です．

リピート・プリフィックス命令は，ストリング命令（MOVSB／MOVSWなど）の直前にだけ置くことができる特別な命令で，CXの値が0になるまで，その直後に置かれたストリング命令を繰り返します．REP命令を使った繰り返しの一般形は，次のようになります．

```
     MOV   CX,カウント
REP  MOVSB
```

それでは，DATA 1：SSSSからnバイトのデータをDATA 2：DDDDにブロック転送（インクリメント・モード）するプログラムを次に示します．

```
MOV   AX,OFFSET   DATA 1 ┐
MOV   DS,AX              ├ DS：SIのセット
MOV   SI,OFFSET   SSSS   ┘
MOV   AX,OFFSET   DATA 2 ┐
MOV   ES,AX              ├ …… ES：DIのセット
MOV   DI,OFFSET   DDDD   ┘
MOV   CX,n ←──────────── ブロック長の設定
```

```
        CLD    ←────────── インクリメント・モードの指定
REP     MOVSB  ←────────── ブロック転送
```

リピート・プリフィックス命令（REP）を用いず，汎用の繰り返し命令 LOOP を用いて上の繰り返しを書くと，

```
ABC：MOVSB
     LOOP   ABC
```

となります．

ブロック転送を行う場合に，データブロックの記憶領域が重なっていなければ，データブロックを先頭から転送しても，後端から転送しても構いませんが，記憶領域が重なっている場合には，どちらからブロック転送するかを考慮しなければなりません．

もし，それを怠ってブロック転送すると，転送中にデータが壊されてしまいます．

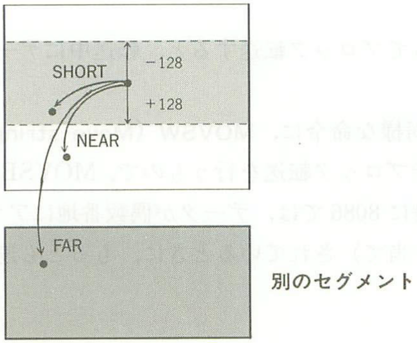
MOVSB 命令と同様な命令に，MOVSW（Move string word）があります．これはワード単位でブロック転送を行うもので，MOVSB 命令よりも高速な転送を行います．一般に 8086 では，データが偶数番地にアラインメント（セグメント開始境界の割り当て）されているときに，もっとも速くアクセスを行うことができます．

# 1.20 無条件分岐命令(JMP)

Jcc 命令は，条件 cc に応じて分岐しましたが，JMP（Jump）命令は，条件に関係なく分岐するものです．JMP 命令の一般形は，

```
JMP    label
```

と書きます．**ラベル（label）の属性**は，**SHORT（±128バイト以内の範囲)**，**NEAR（セグメント内)**，**FAR（セグメント外）**が指定できます．次の図にこれらの関係を示します．



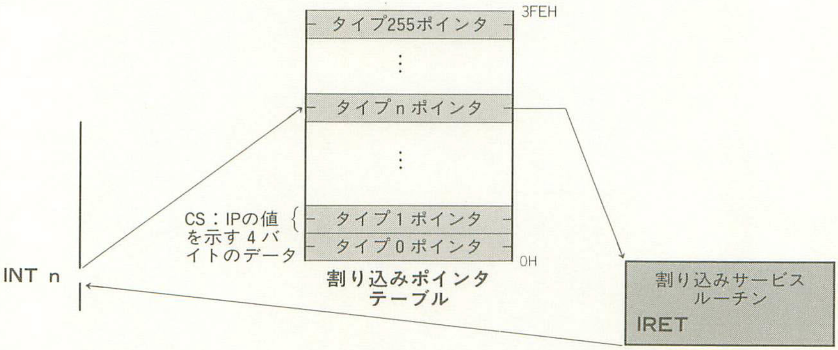

JMP 命令では，このようなラベル指定による直接ジャンプの他にも，レジスタ・ポインタ，メモリ・ポインタを用いた間接ジャンプを行うこともできますが，ここでは省きます．

# 1.21／割り込み命令(INT)

8086における**ソフトウェア割り込み**（Interrupt）は，次のように書きます．

```
INT　n
```

nには，0～255を指定できます．動作はINT　nの割り込みが発生すると，メモリ0番地から設定されている割り込みポインタ・テーブルの第nポインタが参照され，それによって示される割り込みサービスルーチンへ分岐し，割り込み処理終了後，分岐してきた元の位置に復帰します．次の図にこの様子を示します．




割り込みポインタ・テーブルのうち，タイプ0～4（**表1.1**）は特別な割り込みで予約されている他に，各OSによって初期設定されています．たとえば，MS-DOSのシステムコールを行う割り込みタイプは21Hです．したがって，MS-DOSでシステムコールを行うためには次のように命令します．

▼表1.1　8086の予約済割り込み

| タイプ | 割り込みの種類 |
|---|---|
| 0 | 除算エラー(0で割ったとき) |
| 1 | シングルステップ |
| 2 | NMI割り込み |
| 3 | ブレイク・ポイント割り込み |
| 4 | オーバーフロー割り込み |

```
INT　21H
```

# 第2章

## MS-DOSにおける
## アセンブリ言語プログラム開発の基礎

# アセンブリ言語
# プログラム開発

## 本章の目的

マクロアセンブラを使用するには，その文法を覚える必要がありますが，それは第4章，5章において学ぶので，この章ではMS-DOS上でアセンブリ言語によるプログラム開発を行うために必要な基礎事項について説明します．

MS-DOS上での実行可能なファイルは拡張子に.EXEと.COMがつけられたものです．実際にこれらのファイルがどのように異なるのか，これらのアセンブリ言語プログラムのコーディング書式とロード・イメージの違いを元に説明します．さらにこれらのファイルのアセンブル／リンクの概要についても触れることにします．

# 2.1 ／アセンブリ言語プログラム開発ユーティリティ

MS-DOSは16ビットCPU 8086用の**OS（オペレーティング・システム）**であり，この上でアセンブリ言語プログラムを開発するために次の7つのユーティリティが用意されています.

## 1 ■ EDLIN（エディタ）

エディタはアセンブリ言語に限らず,すべてのプログラミング言語のソース・プログラムのテキスト・ファイルを作成するためのユーティリティです.

EDLINは**ラインエディタ**なので，多少使いづらい面もあります．もし，スクリーンエディタを使いたい場合は，**Word Master**などの別売のユーティリティを用意してください.

## 2 ■ MASM（マクロアセンブラ）

16ビットCPU 8086用のマクロアセンブラです.

8ビットCPU 8080／Z80用のマクロ アセンブラ**MACRO-80**との類似性があり，強力なマクロ機能を持っています.

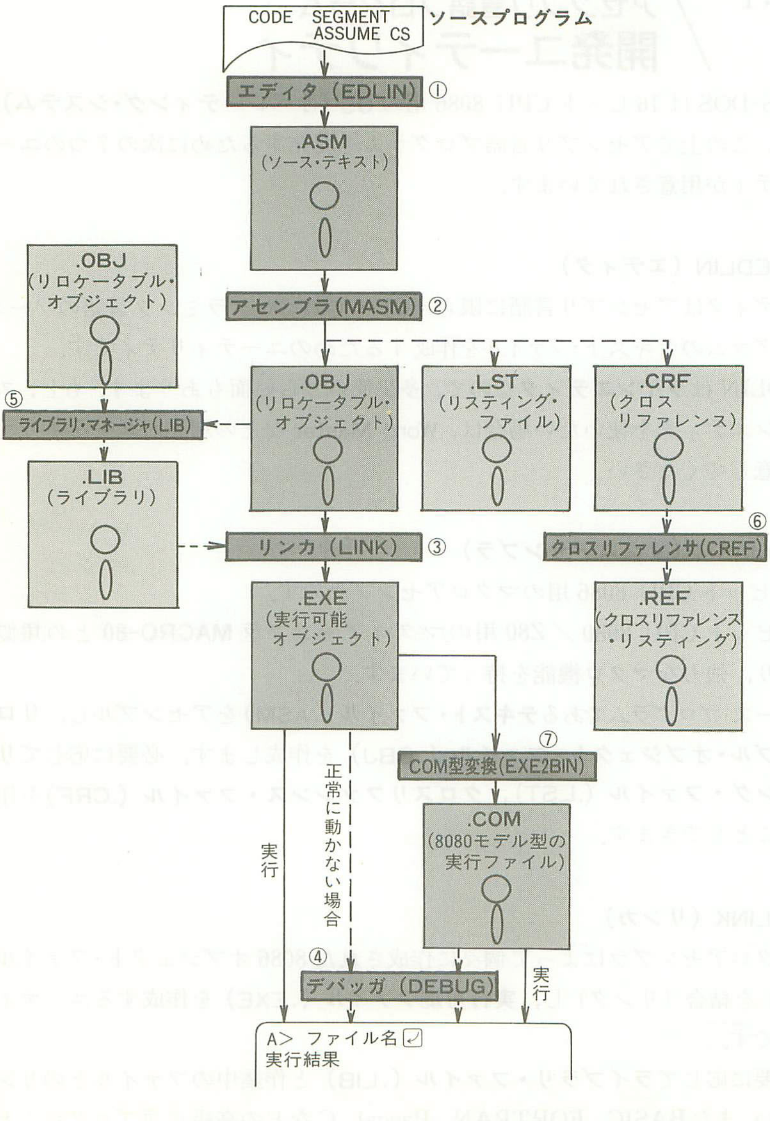
ソース・プログラムである**テキスト・ファイル(.ASM)**をアセンブルし，**リロケータブル・オブジェクト・ファイル（.OBJ）**を作成します．必要に応じて**リスティング・ファイル（.LST），クロスリファレンス・ファイル（.CRF)**を作成することもできます.

## 3 ■ LINK（リンカ）

マクロアセンブラによって個々に作成された8086オブジェクト・ファイル(.OBJ）を結合（リンク）し，**実行可能ファイル（.EXE）**を作成するユーティリティです.

必要に応じて**ライブラリ・ファイル（.LIB）**と作業中のファイルとのリンクも行い,またBASIC，FORTRAN，Pascal，Cなどの高級言語プログラムとのリンクも可能です.

図2.1　アセンブリ言語プログラム開発用ユーティリティの運用形態



**4 ■ DEBUG（デバッガ）**

実行可能なオブジェクト・ファイル（.EXE　または.COM）が正常に動作しなかった場合のテストおよび修正を行うユーティリティです．

**5 ■ LIB（ライブラリ・マネージャ）**

ユーザのライブラリ・ファイル・モジュールの結合・削除・抜き出しを行うユーティリティです．

**6 ■ CREF（クロスリファレンサ）**

ソースプログラム内のシンボル名，およびそれを定義・参照する行番号などに関する**クロスリファレンス・リスト**を作成するユーティリティです．

**7 ■ EXE 2 BIN（.COM ファイル変換ユーティリティ）**

**8080 モデル**（すべてのコード・データを同一のセグメント内に置いている）で作られた.EXE ファイルは，そのままでは正常に動作させることができないので，このユーティリティにより実行可能な**.COM ファイル**に変換します．

これらのユーティリティの運用する場合の関係を**図 2.1** 示します．図中では実線で示した流れが最も基本的な処理手順で，破線の部分は必要に応じて行えばよいものです．

# 2.2 MS-DOSでの2つの実行可能ファイル

MS-DOS における実行可能ファイルはファイルの拡張子が .EXE と .COM のものです.

## ■.EXE モデル

.EXE モデルは, 実行時にコード, データ, スタックなどの領域を各々別の論理セグメントとして割り当て, メモリの記憶領域のどこにでも置いて実行できるリロケータブルなファイルです. MS-DOS におけるもっとも標準的なタイプのものです.

.EXE モデルでのコーディング (コード化) 書式の概要を 図 2.2 に示します.

```
CODE    SEGMENT
        ASSUME    CS:CODE, DS:DATA, SS:STACK
            コード部
CODE    ENDS
DATA    SEGMENT
            データ部
DATA    ENDS
STACK   SEGMENT    STACK
            スタック部
STACK   ENDS
        END
```

図2-2
.EXE モデルの
コーディング書式

## ■.COM モデル

.COM モデルは, 実行時にすべてのコードおよびデータを同一のセグメントに割り当て, その領域を 64 KB 以下に制約しているファイルです. これは, いわゆる 8080 モデルを 8086 上で実現したものであると考えられます.

.COMモデルでのコーディング（コード化）書式の概要を図**2.3**に示します．

```
CODE      SEGMENT
          ASSUME    CS:CODE, DS:CODE, SS:CODE
          ORG   100H
START:
              コード部
              データ部
CODE      ENDS
          END       START ← プログラムの実行開始番地
```

図2-3
.COMモデルの
コーディング書式

## 2.3／.EXEモデルのコーディング書式

.EXEモデルでの基本的なコーディング書式の具体例を60ページ図**2.4**に示し，定形的な基本事項を以下にまとめます．

### 1■セグメントの定義

コードとデータを生成するすべての命令は，いずれかのセグメント内に置かなければなりません．ただし，EQUなどのデータを生成しない擬似命令はセグメントの外に置いても構いません．

セグメントの定義は，SEGMENT～ENDSという擬似命令を用いて，

```
セグメント名  SEGMENT

セグメント名  ENDS
```

とします．図**2.4**の例では，コード・セグメントにCODE，データ・セグメントにDATA，スタック・セグメントにSTACKという名をつけています．

擬似命令SEGMENTの後にはいろいろな指定(アラインメント，コンバイン

タイプ，クラス名)を書くことができます．詳しくは，第5章5.8において説明します．ここでは，スタック・セグメントの場合は，SEGMENT擬似命令の後にSTACKというコンバインタイプを指定する必要があるということを覚えておいてください．

なお，LINK（リンカ）はプログラムのいずれかにスタック・セグメントが置かれていない場合に警告のエラー・メッセージを表示します．

図2.4　.EXEモデルのコーディング例

```
; -----------------------------------------
; *     .EXE model standerd example       *
; -----------------------------------------
;
CR      EQU     0DH
LF      EQU     0AH
;
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA,SS:STACK ←── セグメントの仮定
        MOV     AX,DATA      ;DS set     ┐ DSをDATAセグメントの先頭へ初期設定
        MOV     DS,AX                    ┘
        MOV     AH,9         ;msg disp   ┐
        MOV     DX,OFFSET MSG            │ メッセージの出力（function9）
        INT     21H                      ┘
        MOV     AH,4CH       ;jmp MS-DOS ┐ プログラムを終了し，MS-DOSに戻る
        INT     21H                      ┘
CODE    ENDS

DATA    SEGMENT
MSG     DB      CR,LF
        DB      'Hello MS-DOS world'
        DB      CR,LF,'$'
DATA    ENDS

STACK   SEGMENT STACK        ┐ 32ワードのスタック領域を確保
        DW      32 DUP(0)    │ スタック・ポインタSPはスタックの最下部に設定される
STACK   ENDS                 ┘

        END
```

・このプログラムはスクリーン上に 'Hello MS-DOS world' というメッセージを表示するものです．

### 2 ■セグメントの参照

一般に，

```
JMP     ABC
```

などとしたときの飛び先ラベルABCはコード・セグメント内にあるものとし

て扱われます．これを**コード参照**（CSレジスタとオフセット値による参照）と呼びます．一方，次のように，

```
MOV    BX,ABC
```

としたときの変数ABCはデータ・セグメント内にあるものとして扱われます．これを**データ参照**（DSレジスタとオフセット値による参照）と呼びます．

そこで，**ASSUME擬似命令**によって各セグメント・レジスタ（CS，DS，ES，SS）をどのセグメントに対応させるかを明示しておけば，参照するたびにセグメントを指定するわずらわしさから解放されることになります．

図2.4の例では，

```
ASSUME   CS：CODE，DS：DATA，SS：STACK
```

により，コード参照がコード・セグメントに対して，データ参照がデータ・セグメント，スタック参照がスタック・セグメントに対してそれぞれ行われるということをアセンブラに伝えています．

もし，ASSUMEを使用しなければ，各セグメント内の名前についての参照は，セグメント・レジスタを用いて次のように**オーバーライド**しなければなりません．

```
                ┌──────────── セグメント・レジスタ
                ↓
MOV    BX，DS：ABC
                  ↑
                  セグメント・オーバーライド・プリフィクス
```

### 3■セグメント・レジスタの初期設定

.EXEモデル実行時にCS，DS，ES，SSレジスタはOSによって次のように初期設定されます．

CSレジスタ ……… 先頭セグメントの開始番地

DS，ESレジスタ… 先頭セグメントの前方100Hの領域に確保されたPSP（プログラム・セグメント・プリフィックス）の先頭番地

SSレジスタ…… スタック・セグメントの開始番地

したがって，図2.4の例では，CS，SSの値はプログラム中で初期設定しなく

ても構いませんが，DS の値は最初の PSP（プログラム・セグメント・プリフィックス）の先頭番地を示しているので，次のようにデータ・セグメントの先頭を指すようプログラム中で初期設定しなければなりません．

```
MOV     AX,DATA
MOV     DS,AX
```

**4■プログラムの終了方法**

プログラムを終了し，制御を MS-DOS に戻すには，システムコールのファンクション 4CH を用いて次のように行います．リターン・コードは与えなくても構いません．

```
MOV     AH,4CH
[MOV    AL,リターンコード]
INT     21H
```

## 2.4 / .EXEファイルのロードイメージ

**図 2.4** のプログラムをアセンブル／リンクして作成した.EXE ファイルのロードイメージを**図 2.5** に示します．また，.EXE ファイルを DEBUG（デバッガ）を用いてメモリ上にロードし，各セグメントの内容をダンプしたものを**図 2.6** に示します．

**図 2.5** の中で，CS レジスタはコード・セグメント CODE の先頭，SS レジスタはスタック・セグメント STACK の先頭を示していますが，DS と ES は **PSP（プログラム・セグメント・プリフィックス）**の先頭番地を示しています．

**インストラクション・ポインタ（命令実行の開始ポインタ）IP** は，コード・セグメントのオフセット 0（CS：0）を，スタック・ポインタ SP はスタック・セグメントの最下位を示しています．

**図 2.6** では，CS：IP が 1190：0000 番地ですが，DS の先頭番地 1180 を基準に取れば 1180：0100 番地とも書くこともできることに注意してください．

図2.5　.EXEファイルのロードイメージ

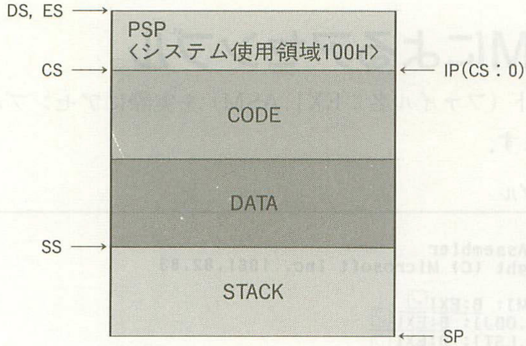



図2.6　DEBUGによる
.EXEファイル・ロードイメージのダンプ

```
A>DEBUG B:EX1.EXE
-R
AX=0000  BX=0000  CX=0070  DX=0000  SP=0040  BP=0000  SI=0000  DI=0000
DS=1180  ES=1180  SS=1193  CS=1190  IP=0000   NV UP DI PL NZ NA PO NC
1190:0000 B89111        MOV     AX,1191
-DDS:0.16F
1180:0000  CD 20 00 40 00 9A F0 FF-0D F0 4A 02 64 0E 78 02   M .@..p..pJ.d.x.
1180:0010  64 0E BA 04 93 0D 64 0E-01 01 01 00 02 FF FF FF   d.:...d.........
1180:0020  FF FF FF FF FF FF FF FF-FF FF FF FF 7D 11 03 2C   ............}...
1180:0030  64 0E 00 00 00 00 00 00-00 00 00 00 00 00 00 00   d...............
1180:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20   M!K.........
1180:0060  20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20           .....
1180:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00           ........
1180:0080  00 0D 42 3A 45 58 31 2E-45 58 45 0D 0D 00 00 00   ..B:EX1.EXE.....
1180:0090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:00A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:0100  B8 91 11 8E D8 B4 09 BA-00 00 CD 21 B4 4C CD 21   8...X4.:..M!4LM!
1180:0110  0D 0A 48 65 6C 6C 6F 20-4D 53 2D 44 4F 53 20 77   ..Hello MS-DOS w
1180:0120  6F 72 6C 64 0D 0A 24 00-00 00 00 00 00 00 00 00   orld..$.........
1180:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

DS, ES → 1180:0000　PSP領域
CS：IP (1190：0000) → 1180:0100
SS → 1180:0140
SS：SP (1193：0040) → 1180:0160
スタック・セグメント
データ・セグメント
コード・セグメント

# 2.5 / MASMによるアセンブル

**図 2.4** のソース・テキスト（ファイル名：EX1.ASM）を実際にアセンブルした具体例を**図 2.7** に示します.

▼図2.7 MASMによるアセンブル

```
A>MASM
The Microsoft MACRO Assembler
Version 1.20, Copyright (C) Microsoft Inc. 1981,82,83

Source filename [.ASM]: B:EX1↵
Object filename [EX1.OBJ]: B:EX1↵
Source listing  [NUL.LST]: B:EX1↵
Cross reference [NUL.CRF]: ↵

Warning Severe
Errors  Errors
0  \    0  \
  警告エラー  致命的エラー
```

MASM 起動後, 次に示す 4 つのファイル名を必要に応じて与えます. ↵キーのみを入力すると, [ ] で囲まれたファイル名が選択されます. NUL はファイルを作らないという意味です.

①ソース・ファイル名

アセンブルするソース・テキスト名です. ファイルの拡張子を省略すると, アセンブラは.ASM とみなします.

②オブジェクト・ファイル名

アセンブルした結果, 生成されるリロケータブル・オブジェクト・ファイルに付けられる名前です. ファイルの拡張子を省略すると, アセンブラは .OBJ とみなします.

③リスティング・ファイル名

アセンブル結果のリスティング・ファイルに付けられる名前です. ファイルの拡張子を省略すると, アセンブラは .LST とみなします.

④クロスリファレンス・ファイル名

クロスリファレンス用ファイルに付けられる名前です．ここでは⏎キーのみの入力なので，ファイルは作成されません．

以上の手順で作成したオブジェクト・ファイル（.OBJ）を図2.8にリスティング・ファイル（.LST）を図2.9に示します．

▼図2.8　リロケータブル・オブジェクト・ファイル(.OBJ)

```
A>DUMP B:EX1.OBJ

Dump Version 2.0

00000000  80 03 00 01 41 3B 96 12-00 00 04 43 4F 44 45 04   ....A;.....CODE.
00000010  44 41 54 41 05 53 54 41-43 4B A0 98 07 00 60 10   DATA.STACK ...`.
00000020  00 02 01 01 ED 98 07 00-60 17 00 03 01 01 E5 98   ........`.......
00000030  07 00 74 40 00 04 01 01-A7 A0 14 00 01 00 00 B8   ..t@....ｱ .....ｸ
00000040  00 00 8E D8 B4 09 BA 00-00 CD 21 B4 4C CD 21 DA   ...ﾘｴ.ｺ..ﾍ!ｴLﾍ!ﾚ
00000050  9C 0B 00 C8 01 04 02 02-C4 08 04 02 02 B4 A0 1B   ...ﾈ....ﾄ....ｴ .
00000060  00 02 00 00 0D 0A 48 65-6C 6C 6F 20 4D 53 2D 44   ......Hello MS-D
00000070  4F 53 20 77 6F 72 6C 64-0D 0A 24 E2 A2 0F 00 03   OS world..$.｢...
00000080  00 00 20 00 01 00 01 00-00 00 02 00 00 28 8A 02   .. ..........(..
00000090  00 00 74 00 00 00 00 00-00 00 00 00 00 00 00 00   ..t.............
```

▼図2.9　リスティング・ファイル(.LST)

```
The Microsoft MACRO Assembler                 10-01-85       PAGE    1-1


                                      ; -----------------------------------
                                      ; *    .EXE model standerd example   *
                                      ; -----------------------------------
                                      ;
 = 000D                               CR        EQU       0DH
 = 000A                               LF        EQU       0AH
                                      ;
0000                                  CODE      SEGMENT
                                                ASSUME    CS:CODE,DS:DATA,SS:STACK
0000  B8 ---- R                                 MOV       AX,DATA         ;DS set
0003  8E D8                                     MOV       DS,AX
0005  B4 09                                     MOV       AH,9            ;msg disp
0007  BA 0000 R                                 MOV       DX,OFFSET MSG
000A  CD 21                                     INT       21H
000C  B4 4C                                     MOV       AH,4CH          ;jmp MS-DOS
000E  CD 21                                     INT       21H
0010                                  CODE      ENDS

0000                                  DATA      SEGMENT
0000  0D 0A                           MSG       DB        CR,LF
0002  48 65 6C 6C 6F 20                         DB        'Hello MS-DOS world'
      4D 53 2D 44 4F 53
      20 77 6F 72 6C 64
```

```
0014  0D 0A 24              DB      CR,LF,'$'
0017                  DATA  ENDS

0000                  STACK SEGMENT STACK
0000      20 [              DW      32 DUP(0)
              0000
                  ]

0040                  STACK ENDS

                            END
```

↑ロケーション・オフセット　↑アセンブルされたコードおよびデータ　↑ソース・テキスト

```
The Microsoft MACRO Assembler          10-01-85   PAGE    Symbols
                                    -1
```

→シンボル(名前)の一覧表

```
Segments and groups:

                N a m e             Size    align   combine class

CODE . . . . . . . . . . . . . .    0010    PARA    NONE
DATA . . . . . . . . . . . . . .    0017    PARA    NONE
STACK. . . . . . . . . . . . . .    0040    PARA.   STACK

Symbols:

                N a m e             Type    Value   Attr

CR . . . . . . . . . . . . . . .    Number  000D
LF . . . . . . . . . . . . . . .    Number  000A
MSG. . . . . . . . . . . . . . .    L BYTE  0000    DATA

Warning Severe
Errors  Errors
0       0
```

# 2.6 LINKによるリンク

図2.8のリロケータブルなオブジェクト・ファイル(ファイル名：EX1.OBJ)を実際にリンクする具体例を図2.10に示します.

▼図2.10 LINKによるリンク

```
A>LINK

   Microsoft Object Linker V2.2(830822)
(C) Copyright 1982, 1983 by Microsoft Inc.

Object Modules [.OBJ]: B:EX1↵
Run File [A:EX1.EXE]: B:EX1↵
List File [NUL.MAP]:↵
Libraries [.LIB]:↵
```

LINK起動後，次に示す4つのファイル名を必要に応じて与えます．⏎キーのみを入力すると，［ ］で 囲まれたファイル名が選択されます．NULはファイル名を作らないという意味です．

①オブジェクト・ファイル名

リンクするオブジェクト・ファイル名です．ファイルの拡張子を省略すると，リンカは.OBJとみなします．

なお，いくつかの.OBJファイルをリンクする場合には，各モジュール名を「＋」で連結します．

②ラン・ファイル名

リンクによって出力される実行可能なファイルに付けられる名前です．ファイルの拡張子を省略すると，リンカは.EXEとみなします．

③リスティング・ファイル名

リンク結果のリスティング・ファイルに付けられる名前です．ファイルの拡張子を省略すると，リンカは.MAPとみなします．

④ライブラリ・ファイル名

.OBJファイルにリンクするライブラリ・ファイルに付けられる名前です．ファイルの拡張子を省略すると，リンカは.LIBとみなします．

以上の手順で作成した実行可能ファイル（.EXE）を図**2.11**に示します．

▼図2.11　実行可能ファイル(.EXE)

```
A>DUMP B:EX1.EXE

Dump Version 2.0
                                            ┌リロケーション情報
                                            ↓
00000000  4D 5A 70 00 02 00 01 00-20 00 00 00 FF FF 03 00   MZp..... ........
00000010  40 00 8B C2 00 00 00 00-20 00 00 00 25 14 00 00   @..ｯ.... ...%...
00000020  01 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

```
00000090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000000A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000000B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000000C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000000D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000000E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000000F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000180  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000190  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000001A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000001B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000001C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000001D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000001E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000001F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000200  B8 01 00 8E D8 B4 09 BA-00 00 CD 21 B4 4C CD 21  ｸ...ﾘｴ.ｺ..ﾍ!ｴLﾍ!
00000210  0D 0A 48 65 6C 6C 6F 20-4D 53 2D 44 4F 53 20 77  ..Hello MS-DOS w
00000220  6F 72 6C 64 0D 0A 24 00-00 00 00 00 00 00 00 00  orld..$.........
00000230  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000240  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000250  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00000260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

機械語コード本体

# 2.7 ／.EXEモデルの簡略化したコーディング書式

リンカ LINK はプログラムの中のどこかにスタック・セグメントがないと警告を発します．しかし、スタック・セグメントがない場合にはスタック・ポインタ SP は 0000 H に初期化されているので、実際のスタックは－1した FFFFH から割り当てられることになります．したがってプログラムの大きさが 64 KB 以下のものならば，警告が発せられていても正常に動作することになります．

そこで，スタック・セグメントを省略したコーディング書式を考えることもできます．その簡略化したコーディング書式を図 **2.12** に示します．

図2.12　簡略化した .EXE モデルのコーディング例

```
; ---------------------------------------
; *     .EXE model simple example       *
; ---------------------------------------
;
CR      EQU     0DH
LF      EQU     0AH
;
CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE
        PUSH    CS              ; DS set     ┐ DSをCODEセグメントの先頭に
        POP     DS                           ┘ 初期設定する
        MOV     AH,9            ;msg disp
        MOV     DX,OFFSET MSG
        INT     21H
        MOV     AH,4CH          ;jmp MS-DOS
        INT     21H

MSG     DB      CR,LF                        ┐
        DB      'Hello MS-DOS world'         │ データもCODEセグメント内に置いた
        DB      CR,LF,'$'                    ┘
CODE    ENDS
        END
```

なお，この例ではコード・セグメントとデータ・セグメントも同一セグメントにしてあります。このため DS の初期設定は，次のようにしても構いません．

```
MOV   AX,CODE
MOV   DS,AX
```

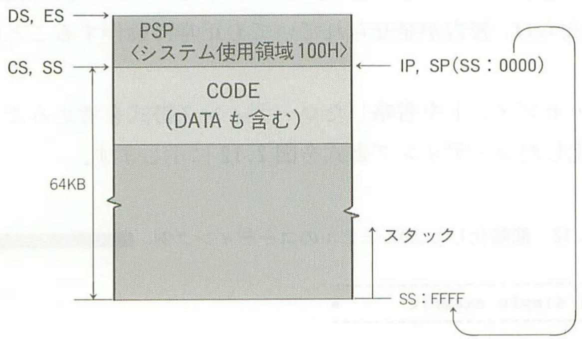
または，CSの値がCODEの先頭に初期設定されているので，

```
PUSH   CS
POP    DS
```

とすることもできます．

図 2.12 の簡略化した.EXEモデルのロードイメージを図 2.13 に示します．

図2.13　簡略化した.EXEファイルのロードイメージ




# 2.8 .COMモデル(8080モデル)のコーディング書式

大きさが64 KB以下のもので，FARコール／ジャンプを行わないプログラムは，リロケート情報が必要ないので，.EXEファイルの先頭にあるヘッダ部を取り除いても正常にロード／実行ができます．このような小規模プログラムを作成するときに.COMモデル（8080モデル）は便利なモデルであるといえます．.COMモデルのコーディング書式を図 2.14 に示します．

図ではCS，DS，SS，ESの4つのセグメント・レジスタはPSPの先頭に，インストラクション・ポインタIPはCS：100に，スタック・ポインタSPはSS：FFFEにそれぞれ初期設定されています．

.COMモデルの利点は，セグメントに関する概念を考えなくてもよいので，

8080／Z80と同じ感覚でプログラミングできることです．

図2.14　.COMモデル(8080)モデルのコーディング例

```
A>TYPE B:EX3.ASM
; -------------------------------------
; *     .COM(8080) model example      *
; -------------------------------------
;
CR      EQU     0DH
LF      EQU     0AH
;
CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,SS:CODE —CS, DS, SSをCODEセグメントに仮定する
        ORG     100H ——————————————————PSP領域の確保
START:  MOV     AH,9            ;msg disp
        MOV     DX,OFFSET MSG
        INT     21H
        MOV     AH,4CH          ;jmp MS-DOS
        INT     21H

MSG     DB      CR,LF
        DB      'Hello MS-DOS world'
        DB      CR,LF,'$'
CODE    ENDS
        END     START ——————————————————プログラム開始番地の指定
```

.COMモデルのコーディング書式の基本的な規則は次の通りです．

1．コード，データ，スタックの各セグメントはすべて同一のセグメントに置かれなければならない．

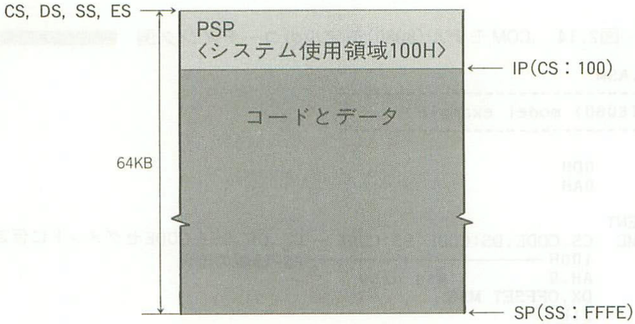
2．プログラムの開始番地がEND擬似命令で定義されなければならない．
この例では，

```
START：MOV    AH,9
                ∫
        END   START
```

により，START番地からプログラムが実行されます．

3．セグメントの先頭にPSP用の100H分の記憶領域を確保しておかなければならない．

図2.15　.COMファイルのロードイメージ




.COMモデルではスタック・セグメントをユーザが定義してないため，リンカはスタック・セグメントがないという警告を発してきますが，それは無視しても構いません．

## 2.9 .EXE ⇨ .COM変換

**図2.14**に示した.COMモデルのファイルをアセンブル／リンクして作った.EXEファイルは，そのままでは正常に実行されません．

そのため，EXE2BINユーティリティにより.EXEファイルを.COMファイルに変換する必要があります．この変換は次の書式で行い，.EXEファイルのヘッダ部と100Hの領域を取り除きます．

```
A>EXE2BIN ␣ filename.EXE ␣ filename.COM
```

上の操作により作られた．COMファイルを**図2.16**に示します．

**図2.16**に示した.COMファイルは，**図2.11**に示した.EXEファイルに比べると相当コンパクトになっていることがわかります．

▼図2.16　.COMファイル

```
A>DUMP B:EX3.COM

Dump Version 2.0
```

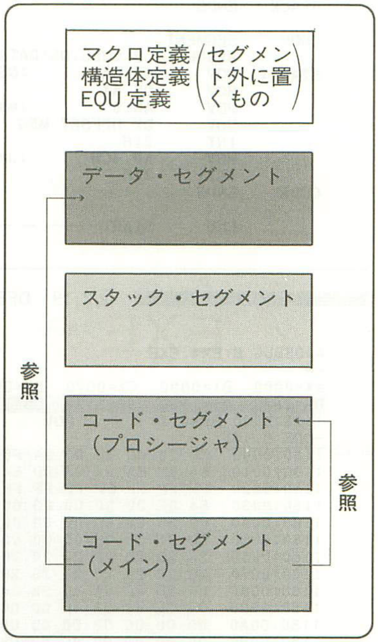
```
00000000  B4 09 BA 0B 01 CD 21 B4-4C CD 21 0D 0A 48 65 6C   ｴ.ｺ..ﾍ!ｴLﾍ!..Hel
00000010  6C 6F 20 4D 53 2D 44 4F-53 20 77 6F 72 6C 64 0D   lo MS-DOS world.
00000020  0A 24 00 00 00 00 00 00-00 00 00 00 00 00 00 00   .$..............
```

# 2.10／前方参照とセグメント配置

**前方参照**とは，自分のコードよりも後に出てくる名前（シンボル）を参照することを指します．前方参照するときに，自分のコードをアセンブル(パス１)している時点ではその名前のタイプや距離を判定することができないので，アセンブラに負担がかかり，最適なアセンブルが行われないこともあります．したがって，前方参照はプログラムの中ではできるだけ少なくしておくことが望ましいプログラミングの方法であるといえます．

また，データやプロシージャのような参照される側のものが定義されているセグメントは，コード・セグメントよりも前に置いた方が前方参照の頻度は少なくなります．

MASMでの一般的なセグメントの配置を**図2.17**に示します．




▲図2.17 一般的なセグメント配置

この配置ではコード・セグメントがセグメントの先頭に来ていないので，

```
END 式
```

によりプログラム開始アドレスを明示しなければなりません．もし怠った場合は，データ・セグメントからプログラムが開始されてしまいます．

**図2.18**に前方参照を避けるためのセグメント配置のコーディング書式を示します．これをアセンブル／リンクした実行可能ファイル(.EXE)をDEBUGを用いてロードしたときのダンプを**図2.19**に示します．

図2.18 前方参照を避けるセグメント配置

```
A>TYPE B:EX4.ASM
; ---------------------------------------
; *     .EXE model standerd example     *
; ---------------------------------------
;
CR        EQU       0DH
LF        EQU       0AH
;
DATA      SEGMENT
MSG       DB        CR,LF
          DB        'Hello MS-DOS world'
          DB        CR,LF,'$'
DATA      ENDS

STACK     SEGMENT STACK
          DW        32 DUP(0)
STACK     ENDS

CODE      SEGMENT
          ASSUME    CS:CODE,DS:DATA,SS:STACK
START:    MOV       AX,DATA        ;DS set
          MOV       DS,AX
          MOV       AH,9           ;msg disp
          MOV       DX,OFFSET MSG
          INT       21H
          MOV       AH,4CH         ;jmp MS-DOS
          INT       21H
CODE      ENDS

          END       START
```

END START ← プログラム開始番地を指定しなければならない

図2.19 DEBUGによるロードイメージ

```
A>DEBUG B:EX4.EXE
-R
AX=0000  BX=0000  CX=0070  DX=0000  SP=0040  BP=0000  SI=0000  DI=0000
DS=1180  ES=1180  SS=1192  CS=1196  IP=0000   NV UP DI PL NZ NA PO NC
1196:0000 B89011          MOV     AX,1190
-DDS:0.16F
1180:0000  CD 20 00 40 00 9A F0 FF-0D F0 4A 02 64 0E 78 02   M .@..p..pJ.d.x.
1180:0010  64 0E BA 04 93 0D 64 0E-01 01 01 00 02 FF FF FF   d.:...d.........
1180:0020  FF FF FF FF FF FF FF FF-FF FF FF FF 7D 11 03 2C   ............}..,
1180:0030  64 0E 00 00 00 00 00 00-00 00 00 00 00 00 00 00   d...............
1180:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20   M!K..........
1180:0060  20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20           .....
1180:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00           ........
1180:0080  00 0D 42 3A 45 58 34 2E-45 58 45 0D 0D 3A 45 58   ..B:EX4.EXE..:EX
1180:0090  34 2E 41 53 4D 0D 00 00-00 00 00 00 00 00 00 00   4.ASM...........
1180:00A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:0100  0D 0A 48 65 6C 6C 6F 20-4D 53 2D 44 4F 53 20 77   ..Hello MS-DOS w
1180:0110  6F 72 6C 64 0D 0A 24 00-00 00 00 00 00 00 00 00   orld..$.........
1180:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:0160  B8 90 11 8E D8 B4 09 BA-00 00 CD 21 B4 4C CD 21   8...X4.:..M!4LM!
```

- DS, ES → 1180:0000
- PSP領域 (1180:0000〜1180:00F0)
- データ・セグメント (1180:0100〜)
- SS → 1180:0120
- スタック・セグメント (1180:0120〜1180:0150)
- コード・セグメント (1180:0160)
- CS：IP(1196：0000) → 1180:0160

# 第3章

# MASMによる8086プログラミング入門

8086
プログラミング

## 本章の目的

第1章，2章で学んだ基礎知識を用いて，MASM による 8086 プログラミングの例題をいくつか解きながらトレーニングしてみることにします．

最初は簡単なものから学習を始めるということにしたため，この章の中ではメモリ・モデルを .COM モデルとしました．したがって実行に際しては，EXE 2 BIN コマンドで .EXE ファイルを .COM ファイルに変換しなければなりません．

一般にアセンブリ言語でプログラムを開発しチェックする場合には，その結果などを画面には表示することができないので，プログラムの確認には DEBUG を用いることにします．この DEBUG の操作方法については，第 10 章で詳しく説明します．

# 3.1 ／ 加減算

**例題1**

DATA 1番地，DATA 2番地，DATA 3番地に格納されているデータ（DATA 1），（DATA 2），（DATA 3）から，（DATA 1）＋（DATA 2）－（DATA 3）を計算し，その結果をRESULT番地に格納せよ．

## プログラム

```
; -------------------------------
; *        加減算プログラム        *
; -------------------------------

CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,SS:CODE
        ORG     100H
START:  MOV     AX,DATA1
        ADD     AX,DATA2 ; (data1)+(data2)
        SUB     AX,DATA3 ;          -(data3)
        MOV     RESULT,AX; 結果の格納
        MOV     AH,4CH   ; return MS-DOS
        INT     21H

DATA1   DW      0005H
DATA2   DW      00FFH
DATA3   DW      0002H
RESULT  DW      0

CODE    ENDS
        END     START
```

### ●結　果

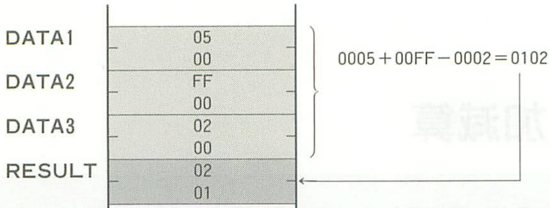
```
A>DEBUG B:DRILL1.COM
-G=CS:100 <---------- プログラムの実行

Program terminated normally
-DCS:100,11F <---------- メモリ内容のダンプ                    プログラム
1180:0100  A1 12 01 03 06 14 01 2B-06 16 01 A3 18 01 B4 4C   !......+...#..4L
1180:0110  CD 21 05 00 FF 00 02 00-02 01 00 00 00 00 00 00   M!..............
                 DATA1                RESULT
```

注）MS-DOS ver. 3.1のSYMDEBの使い方は第12章を参考にしてください．

```
A>SYMDEB  B:DRILL1.COM
-G=CS:100
-D  CS:100,11F
```

【解説】




・セグメント名をCODEとし，CS，DS，SSがコード・セグメントに対応（仮定）している
・ORG 100Hにより，PSP領域を確保している
・END STARTにより，プログラム開始をSTART番地としている

以上のプログラミングは．COMモデルの典型的なものです．

データの定義はDW擬似命令により，

```
DATA1   DW   0005H
```

などとしているので，DATA1はワード属性となります．したがって，

```
MOV     AX,DATA1
```

はワード単位の操作（ワード・オペレーション）になり，レジスタALにDATA1番地の内容を，AHに（DATA1＋1番地）の内容を転送します．

一般に，8086では0005Hなどのワード・データをメモリに格納する場合，下位バイトを先に，上位バイトを後に入れます．これは**逆ワード形式**と呼ばれるインテル社のCPUなどに特有のデータの格納法です．その結果，見かけ上はワード単位のデータをCPUとメモリの間で転送する場合には，上位バイトと下位バイトが入れ換わるように見えます．したがって，05H，00Hという順にメモリには逆の順序に格納される点には今後も注意してください．

加減算はレジスタとメモリ間で行っています．たとえば，

```
ADD     AX,DATA2
```

はレジスタAXとDATA2番地の内容を加算し，その結果をAXに格納します．

# 3.2／メモリの0クリア

**例題2**

SADR 番地から COUNT ワードの範囲のメモリ内容を 0 クリアせよ．SADR と COUNT は EQU で定義すること．

**プログラム**

```
; --------------------------------
; *     メモリー  0  クリアー     *
; --------------------------------

SADR     EQU      0080H    ; 開始アドレス
COUNT    EQU      0080H    ; メモリー・サイズ（byte）

CODE     SEGMENT
         ASSUME   CS:CODE,DS:CODE,SS:CODE
         ORG      100H
START:   MOV      BX,SADR
         MOV      CX,COUNT
ABC:     MOV      BYTE PTR [BX],0
         INC      BX
         LOOP     ABC
         MOV      AH,4CH   ; return MS-DOS
         INT      21H

CODE     ENDS
         END      START
```

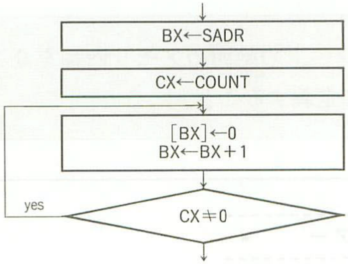
●**結　果**

```
A>DEBUG B:DRILL2.COM
-DCS:80,FF
1180:0080  00 0D 42 3A 44 52 49 4C-4C 32 2E 43 4F 4D 0D 0D   ..B:DRILL2.COM..┐
1180:0090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................│
1180:00A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................│
1180:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................│実
1180:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................│行
1180:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................│前
1180:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................│
1180:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................┘
-G=CS:100

Program terminated normally
-DCS:80,FF
1180:0080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................┐
1180:0090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................│
1180:00A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................│
```

```
1180:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................  実
1180:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................  行
1180:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................  後
1180:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1180:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

**【解説】**




レジスタBXにメモリの先頭アドレス，レジスタCXに繰り返す回数をセットし，CXが0になるまで［BX］←0によって0クリアします．

数値0およびメモリ［BX］は，単独ではバイト型／ワード型の判定ができないため，

```
MOV     [BX],0
```

という表現ではバイトオペレーションなのか，ワードオペレーションか判定ができません．したがって，もしバイトオペレーションならば，

```
MOV     BYTE   PTR [BX],0
```

または，

```
MOV     [BX],BYTE   PTR   0
```

とします．

EQU（Equate）**擬似命令**は名前に値を割り当てます．たとえば，

```
SADR   EQU   0080 H
```

はSADR番地を0080 Hに定義します．SADRなどの番地名を使わないで，

```
MOV     BX,SADR
```

の代わりに，

```
MOV    BX,0080 H
```

としてもよいわけですが，プログラムの中で0080 Hなど具体的な数値を書き込むことはプログラムの修正や移植を考えたときに好ましくありません．そこでEQUにおいてSADRを定義しておいて，プログラムの中でSADRという名前を使用するだけにします．こうすればアドレス値を修正する場合には，プログラム中のSADRには触れずに，EQUによる定義部だけを修正すればよいことになります．EQUなどのデータを生成しない擬似命令は，セグメントの外に置くことができます．

このプログラムでは，CS：80から80 H（128）バイトの領域を0クリアしています．この領域はPSP内のDTA領域というディスクとのデータ授受のためのバッファです．PSP領域については，第11章で説明します．

## 3.3／暗号の解読

**例題3**

DATA番地以後に12文字の暗号文が格納されている．この暗号文を解読せよ．暗号文の文字は，元の文字のアスキーコード＋1（つまりアルファベット順で，元の文字の次の文字）になっている．

**プログラム**

```
; ------------------------------
; *          暗号の解読          *
; ------------------------------

COUNT   EQU     12      ; 暗号の数

CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,SS:CODE
        ORG     100H
START:  MOV     BX,OFFSET DATA←——暗号の先頭位置
        MOV     CX,COUNT ←————暗号の文字数
```

```
ABC:    SUB     BYTE PTR [BX],1←――暗号文字を−1
        INC     BX        ←――――――アドレスを+1
        LOOP    ABC
        MOV     AH,4CH   ; return MS-DOS
        INT     21H

DATA    DB      'HPPE!NPSOJOH'←――暗号

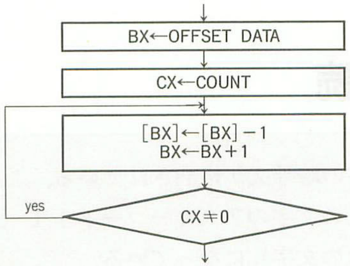
CODE    ENDS
        END     START
```

●結　果

```
A>DEBUG B:DRILL3.COM
-DCS:100,11F
1180:0100  BB 10 01 B9 0C 00 80 2F-01 43 E2 FA B4 4C CD 21   ;..9.../.Cbz4LM!
1180:0110  48 50 50 45 21 4E 50 53-4F 4A 4F 48 00 00 00 00   HPPE!NPSOJOH....
-G=CS:100

Program terminated normally
-DCS:100,11F
1180:0100  BB 10 01 B9 0C 00 80 2F-01 43 E2 FA B4 4C CD 21   ;..9.../.Cbz4LM!
1180:0110  47 4F 4F 44 20 4D 4F 52-4E 49 4E 47 00 00 00 00   GOOD MORNING....
-
               解読された暗号
```

【解説】




レジスタBXに暗号文字が格納されている番地の先頭アドレス（オフセット値），レジスタCXに暗号文の文字数をセットします．次の命令，

```
SUB    BYTE   PTR [BX],1
```

によって，暗号文字が−1され，現在の位置に格納されます．次に，

```
INC     BX
```

により，暗号文字の格納アドレスが1つ先へ進みます．

# 3.4／アスキー変換

**例題4**

1バイトの2進(16進)データを3桁の10進表現で表すために，各桁に対応するアスキー・コードを求め，RESULT 番地以後に格納せよ．

## プログラム

```
; -----------------------------------------
; *     １０進のアスキーコードに変換      *
; -----------------------------------------

CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,SS:CODE
        ORG     100H
START:  MOV     BX,OFFSET RESULT
        MOV     AL,DATA
        MOV     CL,100    ; 第３桁
        CALL    DCMAL
        MOV     CL,10     ; 第２桁
        CALL    DCMAL
        MOV     CL,1      ; 第１桁
        CALL    DCMAL
        MOV     AH,4CH    ; return MS-DOS
        INT     21H

DCMAL:  MOV     DL,2FH    ; サブルーチン
ABC:    INC     DL        ┐
        SUB     AL,CL     │ ALが負になるまでCLが
        JNC     ABC       ┘ 何回引けたかをDLに求める
        MOV     [BX],DL   ┐
        INC     BX        ┘ 結果を格納し，アドレスを＋１する
        ADD     AL,CL     ←─ １回余分に引いた補正
        RET

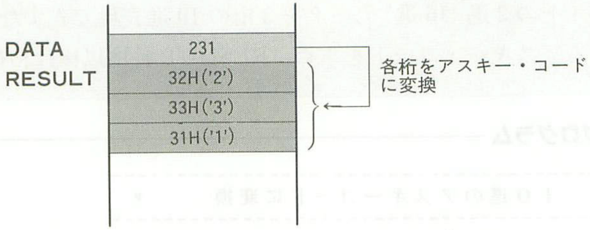
DATA    DB      231
RESULT  DB      3 DUP(?)
CODE    ENDS
        END     START
```

**●結　果**

```
A>DEBUG B:DRILL4.COM
-G=CS:100
```

```
Program terminated normally
-DCS:100,12F
1180:0100  BB 28 01 A0 27 01 B1 64-E8 0E 00 B1 0A E8 09 00   ;(. '.1dh..1.h..
1180:0110  B1 01 E8 04 00 B4 4C CD-21 B2 2F FE C2 2A C1 73   1.h..4LM!2/~B*As
1180:0120  FA 88 17 43 02 C1 C3 E7-32 33 31 00 00 00 00 00   z..C.ACg231.....
-
                                    ↑
                               3桁の10進表現
```

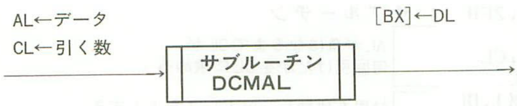
**【解説】**




例として，231（E7 H）の各桁を3桁のアスキー・コードに変換することを考えてみることにします．

まず，第3桁目の2を求めるには，231から100を何回引くことができるかを調べます．その回数が答の2になります．第2桁目の3を求めるには，31から10を何回引くことができるかを調べ，その回数が答の3になります．第3桁は残った1が答になります．

この回数を求める処理を行う部分をサブルーチンDCMALとします．




サブルーチンDCMALを呼び出す（コールする）前に，レジスタALに元のデータを，レジスタCLには引く数（100，10，1など）をそれぞれ与えておきます．サブルーチンDCMALでは，レジスタALからCLを何回引くことができるかを調べます．1回引くたびにレジスタDLが＋1インクリメントされるので，何回引くことができたかはレジスタDLに格納されている回数が示しています．なお，レジスタDLの値を2FH（‘0’－1）に初期設定してあるのは，求めた回数をアスキー・コードに変更するためには30H（‘0’）を後で加えなくてはならないので，あらかじめ0でなく2FHに初期設定しておいて後で加えなくてもよいことにしているのです．

サブルーチンの最後で，

```
ADD    AL,CL
```

を行っているのは，ここではループABCから抜ける判断に減算が負になったらということを条件にしているので，減算処理を1回余計に行ってしまうための対処法です．

このプログラムは，**除算命令DIV**を用いてもう少しコンパクトに書くことができます．それを行うためには，第9章の**19.3**に示してあるマクロbcdを参考にしてください．

## 3.5／ビットパターン

**例題5**

DATA番地に格納されているデータのビットパターンをアスキー・コードでRESULT番地以後に格納せよ．

**プログラム**

```
; ----------------------------------------
; *        ビット・パターンの表示         *
; ----------------------------------------

CODE     SEGMENT
         ASSUME   CS:CODE,DS:CODE,SS:CODE
         ORG      100H
START:   MOV      CX,8
         MOV      AL,DATA
         MOV      BX,OFFSET RESULT
ABC:     SHL      AL,1                    ┐左へ1ビットシフトし，キャリーが
         JNC      BRN1                    ┘出なければBRN1へ飛ぶ
         MOV      BYTE PTR [BX],'1'       ]'1'をセット
BRN1:    INC      BX                      ]アドレスを＋1する
         LOOP     ABC
         MOV      AH,4CH          ; return MS-DOS
         INT      21H

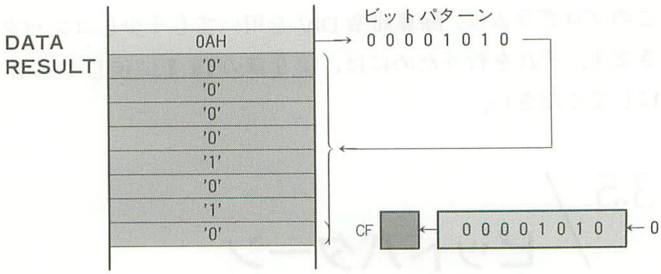
DATA     DB       0AH
RESULT   DB       8 DUP('0')
CODE     ENDS
         END      START
```

## ●結　果

```
A>DEBUG B:DRILL5.COM
-G=CS:100

Program terminated normally
-DCS:100,11F
1180:0100  B9 08 00 A0 17 01 BB 18-01 D0 E0 73 03 C6 07 31   9.. ..:..P`s.F.1
1180:0110  43 E2 F6 B4 4C CD 21 0A-30 30 30 30 31 30 31 30   Cbv4LM!.00001010
                                      ↑
                                  ビットパターン
```

**【解説】**




データ0AHのビットパターンは00001010なので，このビットパターンを上位ビットより取り出して30Hを加え，アスキー・コードの‘0’または‘1’でRESULT番地以後に格納すればよいわけなのですが，実際にはSHL命令を用いてデータを左へ1ビットシフトし，第7ビットをキャリー・フラグCFに格納しています．このときにキャリー・フラグがセットされたか否かを調べることで，第7ビットが0か1かを判定できます．これを8回繰り返せば，8ビット分すべてを調べることができます．

RESULT以後のデータは‘0’に初期設定しておき，キャリー・フラグがセットされたときだけ，次のように命令して該当アドレスを‘1’にします．

```
MOV    BYTE  PTR [BX],'1'
```

MASMにはデータ領域を確保するための専用の擬似命令は用意されてないので，DB／DWとDUPを組み合わせて次のように行います．

```
RESULT  DB  8  DUP('0')
```

これでRESULT以後に8バイトのデータ領域が確保され，‘0’で初期化されます．また，(　　)内に？を書いておけば，初期化は行われません．

# 第4章
# マクロアセンブラの基礎知識

マクロアセンブラ
MASM

## 本章の目的

MS-DOS のマクロアセンブラ（MASM）には，豊富な擬似命令と演算子が用意されています．これらの文法的な説明は第5章において行うことにし，その前段階として，この章では，まずマクロアセンブラ（MASM）を使っていく上で必要な概念についての整理をすることにします．こうした概念を未整理のままにして，いくら多くの命令を覚えようとしてもあまり成果が上がりにくいものだからです．

MASM を使う上では，これから説明する名前やオペランドの書き方といった一般的な規則，セグメントの概念，マクロおよびモジュールの概念などが重要な点となりますので，しっかりと理解してください．

# 4.1／MASMの特徴

MS-DOSのマクロアセンブラ（MASM）は，次のような特徴を持ちます．

**①リロケータブル・アセンブラである**

リンカ（LINK）に入力可能な**リロケータブル（再配置可能）オブジェクト**を出力します．したがって，各ソース・プログラムを独立に作成してアセンブルすることができ，そのオブジェクト・ファイルをリンカによって結合し，さらに大きなプログラムを作成することができます．また，高級言語（BASIC，C，Pascal，Fortranなど）とのリンクを簡単に行うことができます．

**②強力なマクロ機能を持っている**

**マクロ機能**を用いて，たとえばprint（スクリーンへの出力）やinput（キーボードからの入力）などのマクロを定義しておけば，これらのマクロ名を使って，高級言語などで記述する気軽さでアセンブリ言語を記述できます．

**③8086アーキテクチャを生かしたセグメント管理を行っている**

8086のアーキテクチャを生かした**セグメント管理**を行っているので，いくつかの**擬似命令（SEGMENT，ASSUME，GROUPなど）**が用意されており，セグメントに関する柔軟な処理が行えます．

**④データを構造化できる**

単純なデータ定義（バイト／ワード型）の他にDUPを用いた配列，STRUCによる**構造体**など高級言語並みのデータ構造が表現できます．

**⑤プログラムを構造化し易くした**

ある機能単位をPROCにより，**手続き（プロシージャ）**として定義できるので，構造化したプログラムが記述し易くなります．

**⑥型付きアセンブラである**

8086で扱うデータにはバイト型（8ビット）／ワード型（16ビット）の区別やNEAR（セグメント内）／FAR（セグメント外）の区別があります．

どちらの型のデータを扱うかについてを命令に加える場合には，それをニーモニックに含める方法とオペランドに含める方法がありますが，MASMではオペランドに含めているので，データとなるオペランドに型を与えて区別することになります．そのためにデータに付ける名前は，**ラベル**，**変数**，**ナンバ**の3種類に区別され，その使用法と型の定義が重要になります．

**⑦ 8087演算プロセッサのサポート**

8087演算プロセッサの命令もアセンブルすることができます．

## 4.2 ／ソースファイルに関する一般事項

### ■ソース・ファイル名

ディフォルトのファイルの拡張子（ファイルタイプ）として，アセンブラが.ASMを認識するため，ソース・ファイルのファイルタイプは.ASMとするのが一般的ですが，他のファイルタイプでも構いません．

### ■名前に使用できる文字

名前に使用できる文字は次の通りです．

A～Z（a～z）　0～9　?　@　_　$

名前の先頭には0～9の数字を用いてはなりません．名前の長さは何文字であっても構いませんが，先頭31文字だけが判別されます．

### ■数値の表記法

特に指定した場合以外は，10進数として扱われます．10進数以外の進数は，数字の後に次のような文字をつけて区別します．

| 進数 | 表記法 | 例 |
|---|---|---|
| 2 進数 | B | 11001010B |
| 8 進数 | QまたはO | 735Q |
| 10 進数 | Dまたは付けない | 10 |
| 16 進数 | H | 0FFH |

なお，16 進表記の場合には，最初の文字が必ず数字でなくてはならないので，FFH のような値には，その先頭に 0 を付けなければなりません．

**ディフォルト解釈**（特に指定しない場合にアセンブラが自動的に解釈する設定）を 10 進数以外に変更するには，**RADIX 擬似命令**を用います．この擬似命令については第 5 章で説明します．

**■ステートメント（命令文）**

ステートメント（命令文）は，**名前**，**アクション**，**式**，**コメント**の 4 つのフィールドから構成されています．

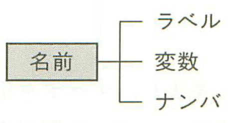
```
START:   MOV        BX,100H          ;program start
  ↑       ↑            ↑                  ↑
 名前   アクション      式              コメント
```

アクション・フィールドには,8086 命令のニーモニック，または MASM の擬似命令が置かれます．

**コメント**は，セミコロン（；）で始まり，そこから行末までがコメントとしてみなされます．また，上記にある 4 つのフィールドのいずれも書かない空白行も認められているので，空白行はプログラムを見やすくするのに役立ちます．

# 4.3／MASMで使う名前

MASM で使う名前は，**ラベル**，**変数**,**ナンバ**の 3 つに分類できます.それぞれ使用法が異なるので，その違いを十分認識しておく必要があります．



■ラベル

ラベルはJMP, CALL, LOOPなどの飛び先に付ける名前で，**コード参照**において用いられます．同一セグメント内のラベルを定義する最も一般的な方法は，次のように名前の後に**コロン**（：）を付ける方法です．

```
        JMP   ABC
               ⋮
   ABC：
   ↑
   ラベル
```

プログラムを構造的に組み立てていくためには，1つのプログラムを，いくつかのある機能を持つ部分のプロシージャとして定義していく方が効果的です．このプロシージャに付ける名前もラベルとして次のように定義されます．

```
CALL  ABC
  ⋮
```

```
ABC      PROC   { NEAR
↑               { FAR
ラベル     ⋮
         RET
         ENDP
```

PROCとCALLが同じセグメントにある場合や2つのセグメントをGROUPやPUBLICの指定により単一の論理セグメントにまとめられる場合は，ラベルABCのタイプはNEARとなり，CALLとPROCが別のセグメントになる場合は，ラベルABCのタイプはFARとなります．

ラベルはこうした2つの定義方法以外にもEXTRN，LABEL擬似命令を用いて定義することができますが，詳しいことは第5章で説明します．

■変数

変数は，ある値の置かれている番地（アドレス）を表し，データを用いる際に参照されます．変数を定義する最も一般的な方法は，DEFINE擬似命令(DB

／DW／DD／DQ／DT）を用いる方法で，

```
        MOV     DL,ABC
           :
    ABC DB      'Hello'
```

とします．変数の後には，コロン（：）が付かないことに注意してください．DEFINE 擬似命令で定義される変数のタイプは次の通りです．

| 擬似命令 | タイプ | サイズ |
|---|---|---|
| DB | BYTE | 1バイト |
| DW | WORD | 2バイト |
| DD | DWORD | 4バイト |
| DQ | QWORD | 8バイト |
| DT | TBYTE | 10バイト |

定義されたこれらのタイプにより暗黙のバイト・オペレーション/ワード・オペレーションが実行されることになります．変数の定義には，これらの他にも，**STRUC，RECORD，LABEL，EXTRN 擬似命令**によっても行われます．これらの命令については第5章を参照してください．

■**ナンバ**

ナンバには **EQU 擬似命令**や**＝擬似命令**によって定義され，データやコードを参照することなく定義される名前です．次のように定義することができます．

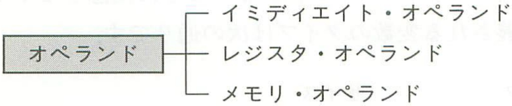
```
    CR  EQU    0DH
    LF  EQU    0AH
           :
        DB     CR, LF, 'hello'
```

EQU 擬似命令と＝擬似命令の違いは，第5章で説明します．

ラベルや変数は，必ずセグメント内で定義されていなければなりませんが，EQU 擬似命令や＝擬似命令で定義された名前はセグメント外にあっても構いません．ナンバの定義は **EXTRN擬似命令**を使っても行うことができます．

# 4.4 / オペランド

オペランドには次の3つのタイプがあります.




## ■イミディエイト・オペランド

イミディエイト（即値）・オペランドは，定義値または EQU および＝擬似命令で定義されたシンボルです．たとえば，

```
        MOV     AX,0 FFFH
```

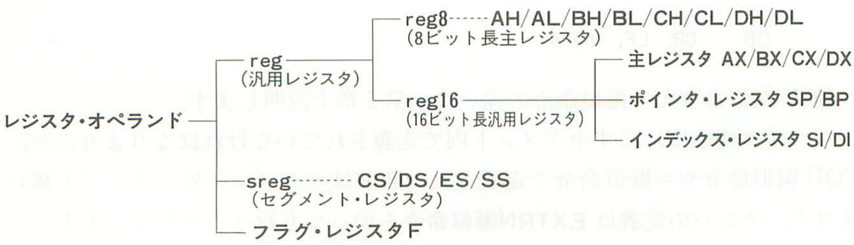
または，

```
SADR    EQU     0 FFH
          ⋮
        MOV     AX,SADR
```

などです．

## ■レジスタ・オペランド

レジスタ名をオペランドとして書く場合，次のように分類することができます．



大きく分けると**reg（汎用レジスタ）**と**sreg（セグメント・レジスタ）**と**フラグ・レジスタF**になります．フラグ・レジスタは一般にオペランドの中には書かずに，PUSHF／POPFなどのようにニーモニックの中に含めて書くので，実際にレジスタ・オペランドとして書くのはregとsregの2種類になります．

sregを指定できる命令は，MOV，PUSH，POPなどに限られており，MOV命令のsregの指定も，regに比べると制限が加わります．

regは**8ビットレジスタ**reg **8**と**16ビットレジスタ**reg **16**に分類され，さらにreg 16は**主レジスタ（AX／BX／CX／DX）**，**ポインタ・レジスタ（SP／BP）**，**インデックス・レジスタ（SI／DI）**に分類できます．

MOV，CMP，ADD，ANDなどの多くの命令では，汎用レジスタregに含まれるどのレジスタも対等にオペランドに書くことができます．しかし，IN，OUT，MUL，DIVなどの命令では，レジスタAXまたはALを中心に行われることや，シフト・ローテイト命令でシフト・ローテイトするビット数を指定するレジスタ・オペランドにはCLだけしか書けないということなど，例外もあるので詳しい内容は**巻末　付録1**を参照してください．

## ■メモリ・オペランド

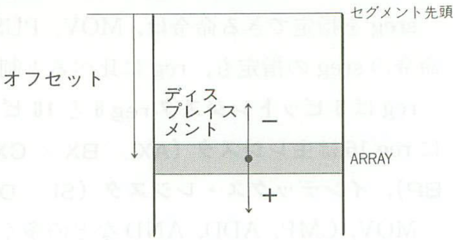
第1章**1.5**ではアドレッシング・モードについて説明しましたが，表現可能なメモリ・アドレッシング・モードを**表4.1**に示します．

●表4.1　メモリ・アドレッシング・モード

| メモリ・アドレッシング・モード | ディフォルト・セグメント・レジスタ |
|---|---|
| $D_{16}$（直接アドレス） | DS |
| ［BX］ | DS |
| ［BX］＋$D_8$／$D_{16}$ | DS |
| ［BX］＋［SI］／［DI］ | DS |
| ［BX］＋［SI］／［DI］＋$D_8$／$D_{16}$ | DS |
| ［BP］＋$D_8$／$D_{16}$ | SS |
| ［BP］＋［SI］／［DI］ | SS |
| ［BP］＋［SI］／［DI］＋$D_8$／$D_{16}$ | SS |
| ［SI］ | DS |
| ［DI］ | DS |
| ［SI］＋$D_8$／$D_{16}$ | DS |
| ［DI］＋$D_8$／$D_{16}$ | DS |

$D_8$は8ビット長のディスプレイスメント，$D_{16}$は16ビット長のディスプレイスメントを示します．

**ディスプレイスメント**とは，オフセットがセグメント先頭からの距離を示すのに対して，セグメント内の任意の位置から他の位置への距離を示します．したがって，オフセットは正の値で表しますが，ディスプレイスメントは正・負両方の値で表されることになります．しかし，普通はディスプレイスメントを表すのに，




```
MOV    AX, ARRAY+5
           ↑── D16
```

または，

```
MOV    AX, [BX] -2
                ↑── D8
```

と書いておけば，後はアセンブラが適当なディスプレイスメントを生成するので，それほど気にしなくても構いません．レジスタ［BP］を用いたメモリ・アドレッシングだけはディフォルト・セグメント・レジスタにSSを用いますが，他はすべてDSを用います．[BP]だけをオペランドにする表現は**表4.1**にはありませんが，もしオペランドに[BP]だけを指定した場合，MASMは[BP]+0と解釈します．

メモリ・オペランドの例を**図4.1**に示します．この例ではARRAYはデータ・セグメントのオフセット0にDWでワード定義されているものとします．

▼図4.1　メモリ・オペランドの例

```
0005  A1 0000 R                    MOV    AX,ARRAY
0008  8B 07                        MOV    AX,[BX]
000A  8B 04                        MOV    AX,[SI]

000C  8B 87 0000 R                 MOV    AX,[ARRAY+BX]     ┐
0010  8B 87 0000 R     ニーモニック──→MOV    AX,ARRAY[BX]      │ ①
0014  8B 87 0000 R                 MOV    AX,ARRAY+[BX]     ┘

0018  8B 84 0000 R ←──機械語コード    MOV    AX,[ARRAY+SI]     ┐
001C  8B 84 0000 R                 MOV    AX,ARRAY[SI]      │ ②
0020  8B 84 0000 R                 MOV    AX,ARRAY+[SI]     ┘

0024  8B 80 0000 R                 MOV    AX,[ARRAY+BX+SI]      ┐
0028  8B 80 0000 R                 MOV    AX,ARRAY+[BX]+[SI]    │
002C  8B 80 0000 R                 MOV    AX,[BX]+ARRAY+[SI]    │ ③
0030  8B 80 0000 R                 MOV    AX,ARRAY[BX][SI]      ┘
```

レジスタ名は［　］で囲まなければなりませんが，それ以外では［　］と＋は同じ意味となります．また，＋でつなぐ順序はどれからでも構いません．したがって，**図 4.1** の①，②，③の中の各表現は同じものであり，それは生成された機械語コードを見比べればわかると思います．

なお，機械語コードの 0000　R は ARRAY のオフセットを示すもので，この例ではオフセットが 0000 であったことを意味します．

# 4.5／ディフォルト・セグメント・レジスタとセグメント・オーバーライド

8086 にあるセグメント・レジスタの使い分けは，暗黙のうちに，命令フェッチには CS レジスタ，スタック操作には SS レジスタ，その他のデータ参照には DS レジスタ（時には，ES レジスタ）ということになっています．これを**ディフォルト・セグメント・レジスタ**と呼びます．メモリを参照する場合に，どのセグメント・レジスタがディフォルトとして設定されているのかを**表4.2**に示します．

●表4.2　ディフォルト・セグメント・レジスタ

| メモリ参照の種類 | | ディフォルト・セグメント・レジスタ | オフセット（レジスタ） |
|---|---|---|---|
| 命令フェッチ | | CS | IP |
| スタック操作 | | SS | SP |
| 一般データの参照* | | DS | 実効アドレス |
| ストリング操作 | ソース側* | DS | SI |
| | デスティネイト側 | ES | DI |
| BS(ベースレジスタ)による参照* | | SS | 実効アドレス |

**表 4.2** の中で＊印の付いているメモリ参照では，ディフォルト・セグメント・レジスタ以外のセグメント・レジスタを指定することができますが，無印のメモリ参照では，それ以外のセグメント・レジスタを指定はできません．

一般にディフォルト・セグメント・レジスタ以外のセグメント・レジスタを指定するときには，**セグメント・オーバーライド**を行わなければなりません．

このため，機械語コードは１バイト余計に生成されます．

まず，**図 4.2** の例を見てください．

▼図4.2　ASSUMEによるデータ・セグメントの仮定

```
0000                      DATA1   SEGMENT
0000  31                  ABC     DB      '1'
0001                      DATA1   ENDS

0000                      DATA2   SEGMENT
0000  32                  XYZ     DB      '2'
0001                      DATA2   ENDS

0000                      CODE    SEGMENT
                                  ASSUME  CS:CODE,DS:DATA1,ES:DATA2
0000  B8 ---- R                   MOV     AX,DATA1
0003  8E D8                       MOV     DS,AX
0005  B8 ---- R                   MOV     AX,DATA2
0008  8E C0                       MOV     ES,AX
000A  8A 16 0000 R                MOV     DL,ABC
000E  26: 8A 36 0000 R            MOV     DH,XYZ
0013                      CODE    ENDS
                                  END
```

↑ES：でのセグメント・オーバーライド

DATA 1 セグメントはDSレジスタを介し，DATA 2 セグメントはESレジスタを介してそれぞれ参照されるということを，**ASSUME 擬似命令**によりアセンブラに指示を与えています．このため，DATA 1 セグメント内の変数ABCへの参照である

```
MOV    DL,ABC
```

という命令は，ディフォルトのDSレジスタを介して行われることになるため，生成される機械語コードはオーバーライドされていません．

一方，DATA 2 セグメント内の変数XYZへの参照である

```
MOV    DH,XYZ
```

という命令ではESレジスタを介して行われるので，生成される機械語コードはオーバーライド（'26：'の部分）されています．

もし，ASSUME擬似命令によるデータ・セグメントの仮定が行われない場合には，**図 4.3** のように，各変数への参照には：演算子を用いてセグメント・オーバーライドしなければなりません．

▼図4.3　：によるセグメント・オーバーライド

```
0000                      DATA1   SEGMENT
0000  31                  ABC     DB      '1'
0001                      DATA1   ENDS
```

```
0000                      DATA2   SEGMENT
0000   32                 XYZ     DB       '2'
0001                      DATA2   ENDS

0000                      CODE    SEGMENT
                                   ASSUME   CS:CODE

0000   B8 ---- R                   MOV      AX,DATA1
0003   8E D8                       MOV      DS,AX
0005   B8 ---- R                   MOV      AX,DATA2
0008   8E C0                       MOV      ES,AX
000A ┌→8A 16 0000 R                MOV      DL,DS:ABC
000E │ 26: 8A 36 0000 R            MOV      DH,ES:XYZ
0013 │                    CODE     ENDS
     │  ↖                          END
     │  ES：でのセグメント・オーバライド
DS：はディフォルトなのでオーバライドされないが，ASSUMEがされていないので
アセンブラにDS：の指示をしておかなければならない
```

# 4.6／参照するセグメント・レジスタの指定（仮定）

各セグメント内のラベルや変数への参照を行うために，8086ではセグメント値とオフセット値という2つの値を用いてます．

セグメント値は**セグメント・レジスタ**（CS，DS，ES，SS）により管理し，オフセット値は一般の命令のオペランドに与えることにしています．したがってアセンブラに対しては，どのセグメント・レジスタを用いてどのセグメント

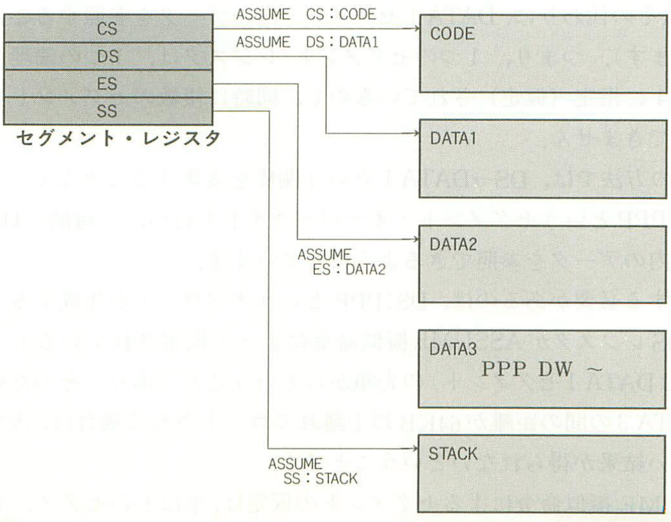



図4.4 セグメントのASSUME擬似命令

を参照するかを明示しておかなければなりません．これを**参照セグメント・レジスタの指定（仮定）**と呼び，ASSUME 擬似命令により行います．

たとえば，**図 4.4** に示したように ASSUME 擬似命令を実行するとします．

これによりコード・セグメントはCSレジスタを用い，DATA 1セグメントはDSレジスタを用い，DATA 2セグメントはESレジスタを用い，スタック・セグメントはSSレジスタを用いて，それぞれ参照することをアセンブラに指示します．

DATA 3セグメントのASSUME 擬似命令は今，実行されていないので，DATA 3セグメント内の変数PPPへDSレジスタを用いて参照したい場合は，次の2つのうちのいずれかの方法で行います．

```
ASSUME   DS：DATA 3
MOV      AX，DATA 3
MOV      DS，AX
MOV      DX，PPP
```

または，

```
MOV      DX，DS：PPP
```

前者の方法ではDS→DATA 1という関係をDS→DATA 3に変更したために，これ以後はDSレジスタを用いてDATA 3セグメント内のデータを参照することができます(その代わりに，DATA 1セグメント内のデータを参照することはできなくなります)．つまり，1つのセグメント・レジスタは，1つの論理セグメントに1対1に指定（仮定）されているので，同時に複数のセグメントを仮定することはできません．

一方，後者の方法では，DS→DATA 1という関係を変更することなく，オペランドでDS:PPPというセグメント・オーバーライドを行い，一時的にDATA 3セグメント内のデータを参照できるようにしています．

ここで注意する必要があるのは，DS:PPPというオペランドが生成するオフセット値は，DSレジスタがASSUME擬似命令によって仮定されているセグメント(ここではDATA 1セグメント)の先頭からということであり，そのためにDATA 1とDATA 3の間の距離が64KB以上離れてロードされる場合は，後者の方法では正しい結果が得られないということです．

なお，ASSUME 擬似命令によるセグメントの仮定は，単にどのセグメント・レ

ジスタを用いて，何セグメントを参照するかをアセンブラに指示するだけのものであり，セグメント・レジスタをASSUME擬似命令を用いてセグメントの先頭に初期設定する動作までは含んでいません．したがって，セグメント・レジスタの初期設定はプログラマの責任として行われなければなりません．

アセンブラは，ASSUME 擬似命令，またはセグメント・オーバーライド（DS: など）により，オペランドに与えられた変数を参照する際のセグメント・レジスタが何であるかを知ることができ，そのレジスタがディフォルトされているもの以外ならば，アセンブルする際にセグメント・オーバーライドを機械語コードの前に挿入します．

## 4.7／セグメントの定義

コードとデータを生成するすべての命令は，いずれかのセグメント内に置かなければなりません．このセグメントを定義するのが**SEGMENT～ENDS 擬似命令**でした．

セグメントの数が複数になった場合は，セグメントのロード順序，セグメントの組み合わせタイプ，セグメントの開始境界の指定などをSEGMENT～ENDS 擬似命令で指定する必要が生じてきます．ここでは，セグメントを定義する場合にどういうことが行われているかなど，セグメントの定義の概念について説明します．

なお，SEGMEMT 擬似命令の書式は次の通りです．

```
セグメント名 SEGMENT ［アラインメント］［コンバインタイプ］［'クラス名'］
        ⋮
セグメント名  ENDS
```

### ■組み合わせ（コンバイン）タイプ

セグメントを1つの論理セグメントにするか，まったく独立したものにするかといったセグメントの組み合わせ方を**コンバインタイプ**と呼び，次の6種類があります．

・無指定（PRIVATE）
・PUBLIC
・STACK
・COMMON
・MEMORY
・AT（式）

COMMON，MEMORY，AT（式）についてはそれほど頻繁に使わないものなので，ここでは説明を省略します．

**PRIVATE 指定**されたセグメントは各々が独立しており，独自の**セグメント・ベース**を持ちます．したがって各セグメントのオフセットは，各セグメント・ベースを基準としたものになります．このようすを**図 4.5** に示します．

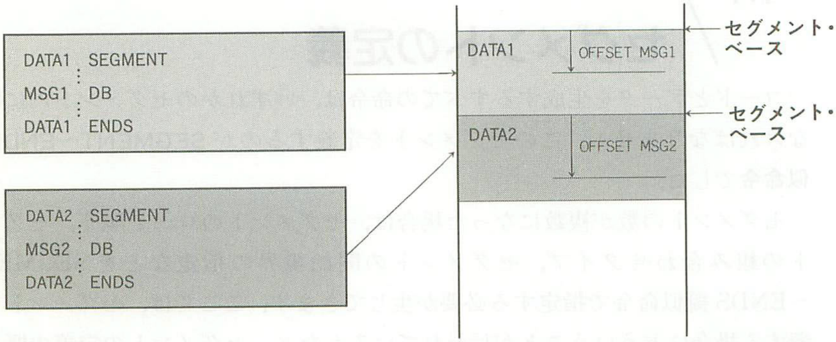



図4.5 PRIVATE（無指定）

**PUBLIC 指定**された論理セグメントは，同じクラス名，または同じセグメント名を持つすべてのセグメントが連結されて１つの論理セグメントとして扱われます．したがって，各セグメントのオフセットは連結された１つの論理セグメントのベースを基準としたものになります．このようすを**図 4.6** に示します．

**STACK 指定**は，PUBLIC と同様に同じクラス名，または同じセグメント名を持つすべてのセグメントを連結しますが，スタック・ポインタ SP の値をスタックの底に初期設定する点だけが異なります．この様子を**図 4.7** に示します．

スタック・セグメントに STACK のコンバインタイプを指定しておけば，スタック・ポインタ SP の初期設定をプログラム中で行わなくてもよいことになります．

図4.6　PUBLIC指定

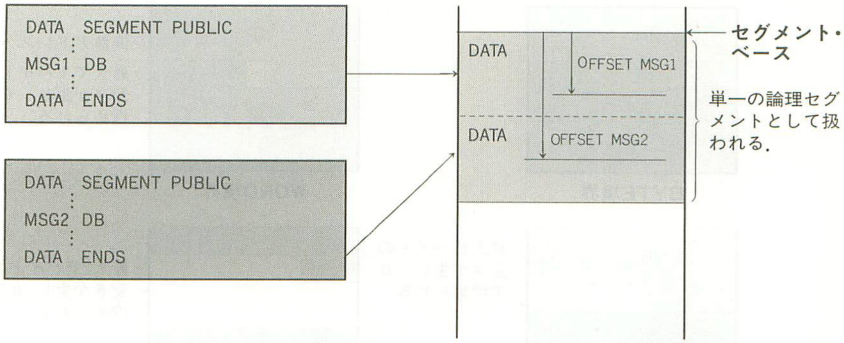
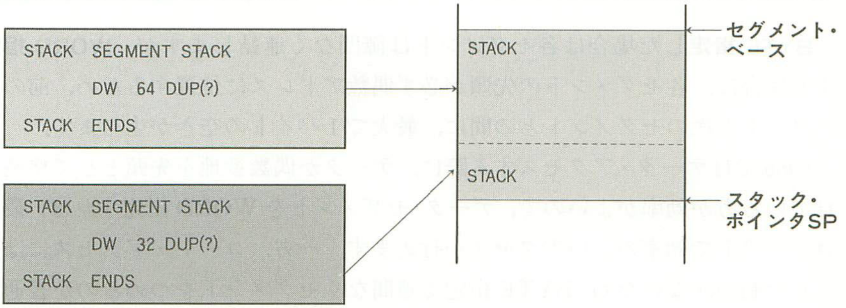



図4.7　STACK指定




## ■セグメント開始境界（アラインメント）

セグメントの開始境界（アラインメント）をどのように設定するかは，次の4種類の指定の方法があります．

- ・BYTE（バイト境界：1バイト）
- ・WORD（ワード境界：2バイト）
- ・PARA（パラグラフ境界：16バイト）
- ・PAGE（ページ境界：256バイト）

これらの指定を省略すると，アセンブラはPARA指定として解釈します．各

指定によるアラインメントの様子を図 4.8 に示します．

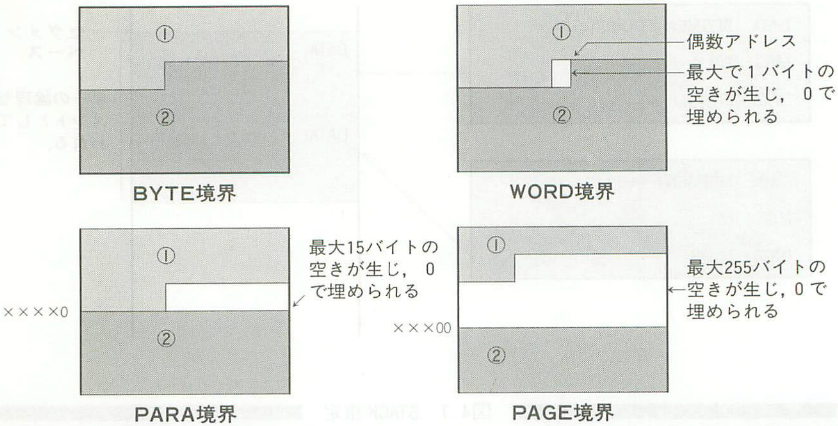



図4.8 アラインメント

**BYTE 指定**した場合は各セグメントは隙間なく連結しますが，**WORD 指定**する場合は，各セグメントの先頭が必ず偶数アドレスに位置するため，前のセグメントと次のセグメントとの間に，最大で1バイトの空きが生じます．

8086 ではデータ・アクセスする際に，データが偶数番地を先頭として格納されている方が効率がよいので，データ・セグメントを WORD 指定すれば，最もコンパクトで効率のよいアクセスが行えます．一方，コード・アクセスにはこうした制約がないため，BYTE 指定で隙間なくセグメントをつめるのが普通です．

**PARA 指定**では，次のセグメントの開始位置が 16 の倍数(つまり，下位 4 ビットが 0）になる番地から始まるよう調整されます．

**PAGE 指定**では，256 の倍数(つまり，下位 8 ビットが 0)になる番地から始まるよう調整されます．

**■セグメントのロード順序**

セグメントのロード順序を設定するパラメータには，セグメント名とクラス名があります．セグメント名はアセンブル時に，クラス名はリンク時にセグメントの配置を指定します．

MASM は,ソース・プログラムの中にセグメント名があれば，それらをすべて

同一の属性を持つひとかたまりのセグメントとしてまとめます．したがって，1つのソース・プログラムの中では,同じセグメント名のセグメントは，その属性（コンバインタイプ，アライン，クラス名）はすべて同じでなければなりません．

リンカでは，リンクするモジュールの中の同じクラス名を持つセグメントを，指定されたコンバインタイプやアラインメントにしたがって連結します．

セグメントのロード順序に関する具体例を図4.9に示します．

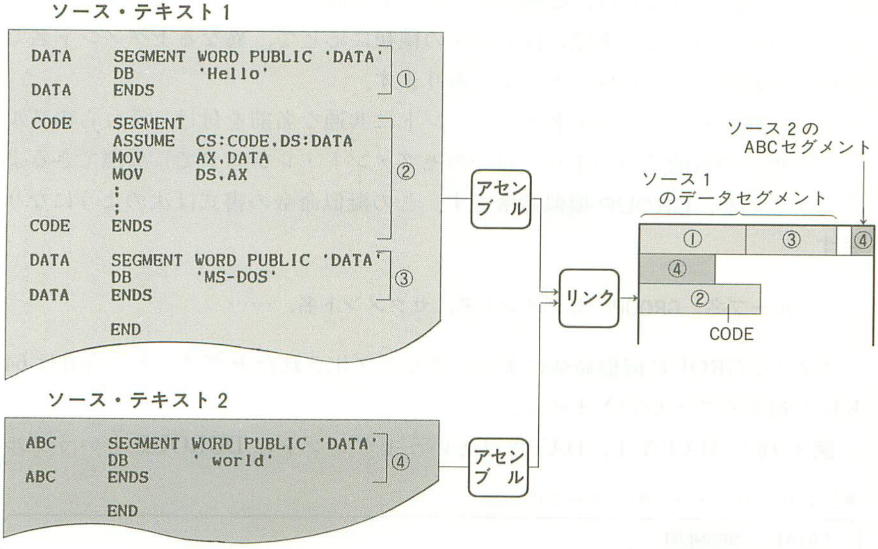



図4.9　セグメントのロード順序

ソースプログラム1のデータ・セグメント①，③は，WORD PUBLIC 'DATA' 属性を持つ1つのセグメントにまとめられます．その後，ソースプログラム2のABCセグメント④が同一のクラス名 'DATA' を持つものとして連結されます．このときのアラインメントはWORD指定されているので，サイズによって1バイトの空きが生じます．

# 4.8 / セグメントのグループ化

異なるセグメント名のセグメントでは，単一のセグメント・レジスタを用いたASSUME擬似命令で，どのセグメントを参照するかを指定できないので，セグメントが移るたびに指定し直すか，セグメント・オーバーライドする必要があります．それならば，必要なセグメントを同じセグメント名でまとめておけばよいかというと，実際にはデータの種類に応じて，異なるセグメント名で管理しておきたいということがよくあります．

そこで異なるセグメント名のセグメントに共通な名前を付け，それらのグループを単一の名前で（つまり，単一のセグメント・レジスタで）参照できるようにするのが，**GROUP 擬似命令**です．この擬似命令の書式は次のようになります．

グループ名　GROUP　セグメント名，セグメント名，……

ただし，GROUP 擬似命令によってグループ化されたセグメントの合計は64 KBを超えることができません．

図4.10にDATA 1，DATA 2というセグメントにDGROUPというグル

▼図4.10　セグメントのグループ化

```
DATA1    SEGMENT
ABC      DB 'Hello'
DATA1    ENDS

DATA2    SEGMENT
XYZ      DB 'MS-DOS'
DATA2    ENDS

CODE     SEGMENT
DGROUP   GROUP    DATA1,DATA2←———DATA1, DATA2セグメントをDGROUPとする
         ASSUME   CS:CODE,DS:DGROUP←—DGROUPはDSを介してアクセスできる
         MOV      AX,DGROUP ┐
         MOV      DS,AX     ┘←————DSをGDROUP先頭に初期設定
         :
         MOV      AL,XYZ
         MOV      DX,OFFSET XYZ
         MOV      DX,OFFSET DGROUP:XYZ
         LEA      DX,XYZ
         :
CODE     ENDS

         END
```

ープ名を付け，DGROUP 内のセグメントは DS レジスタを介して参照できるようにした例を示します．

GROUP によりグループ化されたセグメントは単一のセグメント・ベースを持つため，

```
MOV    AX,DGROUP
MOV    DS,AX
```

としてレジスタ DS を DGROUP の先頭（セグメント・ベース）に設定して，

```
MOV    AL,XYZ
```

とすればDATA 2 セグメント内の変数XYZの内容を参照することができます．

一方，OFFSET 演算子は，変数が置かれている論理セグメントの先頭からのオフセット値を求めるので，

```
MOV    DX,OFFSET   XYZ
```

という命令文では，正しい DGROUP 先頭からのオフセット値を求められません．正しいオフセット値を求めるには，DGROUP：を用いてセグメント・オーバーライドし，

```
MOV    DX,OFFSET   DGROUP : YXZ
```

としなければなりません．このようすを図 4.11 に示します．

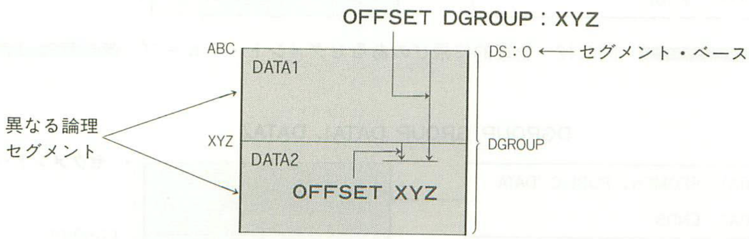



図4.11　グループ内のオフセット

なお，次のように OFFSET 演算子を用いないで，直接 LEA 命令によって変数 XYZ のオフセット値を得る方法を採れば，DGROUP によるセグメント・オーバーライドは必要ありません．

```
LEA     DX, XYZ
```

ここで注意することは，GROUP 命令自体にはセグメントの配置を指示する機能は含まれてはおらず，セグメントの配置はあくまでも，定義されているセグメントの属性（コンバインタイプ，アラインメント，クラス名）で決定されるということです．したがって**図 4.10** のセグメントの配置は，DATA 1 セグメント，CODE セグメント，DATA 2 セグメントの順序になっていますが，DATA 1，DATA 2 セグメントを DGROUP にグループ化すると，その配置は**図 4.12** のようになります．

最初に，GROUP 擬似命令によってグループ化されたセグメントの合計は 64 KB を超えることができないといいましたが，**図 4.12** のようなセグメント間の空きも合計に含まれるので，注意してください．

また，DATA 1，DATA 2 セグメントが同一のクラス名を持てば，**図 4.13** に示すような 1 つの論理セグメントにまとめられます．

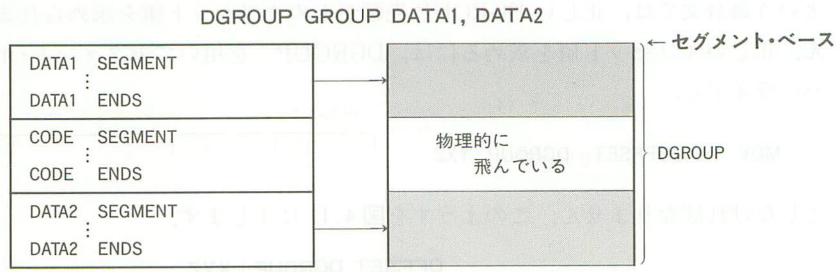



図4.12　物理的に飛びのあるセグメント・グループ

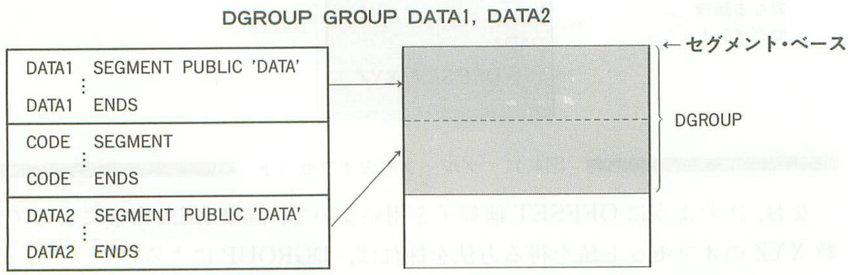



図4.13　同一のクラス名を持つセグメントのグループ化

ある種類のコンパイラでは，コンパイラがクラス名を定義することができるので，リンクする場合には同一のクラス名を用いてグループ化しなければなりません．たとえば，Lattice C ではプログラムのグループには‘PROG’，データのグループには‘DATA’という決められたクラス名を付けるという規則になっています．

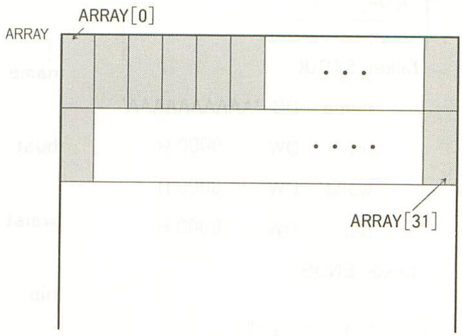
## 4.9／データの定義

単純なデータ定義は，DB/DW などの**擬似命令**を用いて行われますが，MASM では，さらに**配列**や**構造体**といった高級言語並みの複雑な構造を持ったデータを定義することができます．

### ■配列

CP/M-80におけるRB命令などのような複数の領域を確保するための特別な擬似命令はMASMに用意はされてなく，この操作は，**DB/DW などの擬似命令と DUP 命令**により行います．

```
ARRAY    DB   32   DUP (?)
```




により，ARRAY 番地以後に 32 バイトのバイト型領域を確保します．? 指定では，領域の初期設定は行いませんが，DUP (‘A’) などとすると，‘A’でその領域を初期設定します．

この配列の 5 番目の要素に 1 をロードするには，

```
MOV    ARRAY[5],1
```

とします．また，インデックスレジスタ SI を用いるならば，

```
MOV    SI, 5
MOV    ARRAY[SI],1
```

とします.

**■構造体**

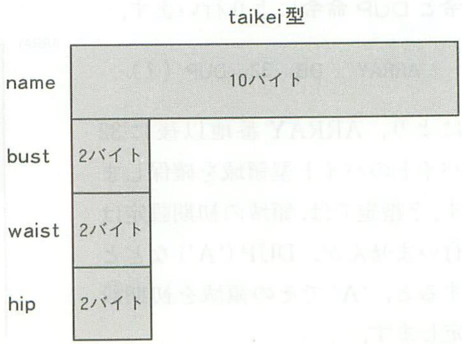
たとえば，次のような個人のデータがあったとします.

| 名　前 | bust | waist | hip |
|---|---|---|---|
| Ann | 80 | 65 | 79 |
| Nancy | 76 | 63 | 76 |
| ⋮ | | | |

このようなデータは，データの型が異なっていても1人分のデータ（名前，bust，waist，hip）をひとまとめにして扱った方が便利です．そこで，個々のデータをまとめたものを1つのデータ型にするというのが**構造体**の考え方です．構成する各データの要素が同じ型であるならば，配列でも代用できますが，構造体では各データの要素が異なっていても用いられるので便利です．

構造体を宣言するには，**STRUC 擬似命令**を用います．上の個人のデータを例とすると，

```
taikei STRUC
    name  DB  'AAAAAAAAAA'
    bust  DW     0000 H
    waist DW     0000 H
    hip   DW     0000 H
taikei ENDS
```




のように宣言します.

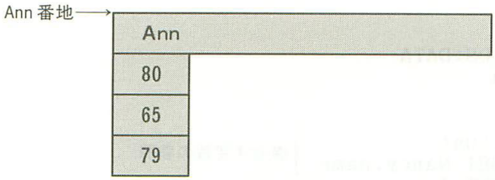
このtaikeiを**タグ**(tag)，name，bust，waist，hipを**メンバ**（member）と呼びます．ここでの宣言では，構造体としてtaikeiというデータ型の宣言を行っているだけなので，定義による領域の確保は行ってはいません．データを定義するにはタグtaikeiを用いて，

```
Ann   taikei  <'Ann', 80, 65, 79>
```

とします．構造体のメンバを初期設定するには，各メンバの要素を書いて，<>で囲むだけで済みます．初期設定しないメンバには何も書かずに**コンマ（,）**だけを書きます．

なお，STRUC擬似命令による宣言のときに，DUPによりメンバを定義してあれば，そのメンバは初期設定できません．詳しい内容は第5章で説明します．

これで次のように，Ann番地以後に領域が確保され，データが初期設定されます．




構造体の各メンバを参照するには，ピリオド（.）を用いて，

```
変数名 .メンバ
```

とします．この変数名とはAnnなどを指し，たとえばAnnのbustデータをレジスタAXにロードするには次のように，

```
MOV    AX,Ann.bust
```

と命令し，Annのnameデータの先頭オフセットをレジスタBXにロードするには，

```
MOV    BX,OFFSET  Ann.name
```

と命令します．

一般に，STRUC擬似命令による宣言はセグメント外で行われ，構造体の定義はデータ・セグメント内で行われます．このようすを112ページの図4.14に示します．

▼図4.14　構造体の定義

```
taikei  STRUC
    name  DB 'AAAAAAAAAA'
    bust  DW 0000H
    waist DW 0000H
    hip   DW 0000H
taikei  ENDS

DATA    SEGMENT
Ann     taikei <'Ann',80,65,79>
Nancy   taikei <'Nancy',76,63,76>
DATA    ENDS

CODE    SEGMENT
        ASSUME CS:CODE,DS:DATA
        MOV     AX,DATA
        MOV     DS,AX

        MOV     AX,Ann.bust
        MOV     BX,OFFSET Nancy.name
        MOV     BX,OFFSET Ann
        MOV     AX,[BX].hip

CODE    ENDS

        END
```

タグ taikei の宣言（taikei STRUC ～ taikei ENDS）

構造体変数Ann, Nancyの定義（DATA SEGMENT ～ DATA ENDS）

構造体変数の参照（MOV AX,Ann.bust ～ MOV BX,OFFSET Nancy.name）

# 4.10 ／ マクロ

MASM は**マクロアセンブラ**と呼ばれています．したがって，MASM を使う際に他の機能は差し置いても，**マクロ機能**を使わないことには，MASM を利用する価値がないといえます．また，マクロ機能はそれくらい便利な機能です．

マクロ機能をひと言で説明するとするならば，アセンブラの命令をサブルーチン化したものというイメージになります．プログラムを作成していくと，その過程で同じ処理がプログラムのあちこちに出てくることがよくあります．このような場合には，その重複する処理をサブルーチンにしておいて，サブルーチンコールを行います．

同様にマクロ機能も，重複する処理をマクロ定義しておいてから，必要なときにマクロコールするというもので，サブルーチンと似ています．

マクロは一般に，次のように定義します．

```
名前  MACRO   [仮引数，…]

      マクロ本体プログラム

      ENDM
```

このマクロを呼ぶには，

```
名前  [実引数，…]
```

とします．

では，114ページの**図 4.15** で具体例を見ることにします．この例では DISP というマクロを定義しています．このマクロには，2 つの引数 COUNT と ASC があり，ASC で与えられた文字を COUNT の回数だけスクリーンに出力するものです．

▼図4.15　マクロの例

```
A>TYPE B:MACRO.ASM
; ----------------------------------
; *       macro example program     *
; ----------------------------------
                                  ┌─仮引数
DISP      MACRO     COUNT,ASC
          LOCAL     START ────ラベルSTARTをLOCAL指定
          MOV       CX,COUNT
          MOV       DL,'&ASC'
START:    MOV       AH,2
          INT       21H
          LOOP      START
          ENDM

CODE      SEGMENT
          ASSUME    CS:CODE,DS:CODE
          PUSH      CS     ┌─実引数
          POP       DS
          DISP      10,V ───マクロコール
          DISP      5,* ───マクロコール
          MOV       AH,4CH
          INT       21H
CODE      ENDS
          END
```

このマクロをコールするには，たとえば，

```
DISP   10, V
```

とします．これにより，COUNTに10が渡され，ASCにはVが渡されます．

**図4.15**のソースリストをアセンブルした結果を**図4.16**に示しますので，マクロの機械語コードがどのように生成されるかを見てください．

ソースプログラムのマクロ定義部は，機械語コードには落されません．その代わりにメインルーチンでマクロを呼び出した位置にマクロ定義部（マクロ名，マクロ本体プログラム）が挿入されることになります．

一般にマクロアセンブラは，マクロを呼び出した位置に，そのマクロ定義の本体を置き換えてからアセンブルを行います．マクロをその**定義**本体で置き換えることを**マクロ展開**と呼びます．

```
DISP   10,V  ──────────→  MOV   CX,10  ──────────→  B9   000A
             〈マクロ展開〉                〈アセンブル〉
                          MOV   DL,'V'               B2   56
                            ⋮                          ⋮
```

▼図4.16　マクロ展開

```
                                          ; ---------------------------------
                                          ; *      macro example program      *
                                          ; ---------------------------------

                                  DISP    MACRO   COUNT,ASC
                                          LOCAL   START
                                          MOV     CX,COUNT
                                          MOV     DL,'&ASC'
                                  START:  MOV     AH,2
                                          INT     21H
                                          LOOP    START
                                          ENDM

0000                              CODE    SEGMENT
                                          ASSUME  CS:CODE,DS:CODE
0000   0E                                 PUSH    CS
0001   1F                                 POP     DS
                                          DISP    10,V
0002   B9 000A             +              MOV     CX,10
0005   B2 56               +              MOV     DL,'V'
0007   B4 02               + ??0000:      MOV     AH,2
0009   CD 21               +              INT     21H
000B   E2 FA               +              LOOP    ??0000
                                          DISP    5,*
000D   B9 0005             +              MOV     CX,5
0010   B2 2A               +              MOV     DL,'*'
0012   B4 02               + ??0001:      MOV     AH,2
0014   CD 21               +              INT     21H
0016   E2 FA               +              LOOP    ??0001
0018   B4 4C                              MOV     AH,4CH
001A   CD 21                              INT     21H
001C                              CODE    ENDS
                                          END
```

（注記）マクロ展開されたことを示す（＋）
（注記）DISP　10，V のマクロ展開
（注記）'&ASC'が'＊'に置き換った
（注記）LOCAL 指定により付けられたラベル（??0000:，??0001:）

このマクロ DISP については，説明しておかなければならないことが2つあります．

そのひとつは，マクロ定義内にラベルがある場合です．メインルーチンからは DISP のマクロが2回呼び出されるので，マクロ展開されたときに START というラベルが2個所に現れます．したがってラベルの二重定義とみなされてエラーとなります．これを防ぐには **LOCAL 擬似命令**を用いて，

```
LOCAL    START
```

とします．これによって，マクロ展開するときに START というラベルは「??」で始まる ??0000〜??FFFF というアセンブラが定めるラベルに付け換えられます．

もうひとつ注意するところは，‘ ’内に書いた引数の扱い方です．一般に‘ ’で囲まれたものは単なる文字列ですから，‘ASC’としただけではアセンブラは引数として認めてくれません．したがって‘ ’内に引数を書く場合は，それが単に

文字列ではなく引数であることを明示するために，引数の先頭に＆演算子を付け，'&ASC' とします．引数 ASC に V が渡されれば，'&ASC' は 'V' と展開されます．

さて，ここでサブルーチンコールとマクロコールの違いがわかってきたと思えるので，少し整理をしてみたいと思います．

サブルーチンコールでは，必ず CALL という命令が機械語コードに置き換えられてその位置に置かれます．サブルーチン本体がその位置に置き換えられるということは決してありません．つまり，サブルーチンコールでは，プログラム実行時に CALL 命令でサブルーチンが置いてある場所へジャンプし，RET 命令によって元の位置へ戻ってくるという動作を行い，サブルーチン自体は置かれた位置から移動しません．したがって，メインルーチンからサブルーチンへジャンプしたり，リターンするので，それだけ実行時間にオーバーヘッドが生じます．

一方，マクロコールではマクロが呼び出された位置にマクロ本体が組み込まれるため，どこへもジャンプする必要はない代わりに，呼び出したマクロはすべてマクロ本体に置き換えられるため，生成された機械語コードでは重複したコードが存在し，余分にメモリを消費します．その点サブルーチンコールではメモリの消費は少なくて済みます．

このような展開の仕方の違いだけでなく，マクロには強力な記述能力があり，ユーザ定義の**マクロライブラリ**を作成できるという長所があります．これにより，アセンブラでありながらも高級言語で記述するような感覚でプログラミングをすることができます．

具体的なマクロ機能の利用例は第 12 章で説明します．

# 4.11／プロシージャの定義と参照

**プロシージャ**（サブルーチン）を定義するには，**PROC擬似命令**を用いて，

```
名前  PROC  {NEAR
             FAR}

      サブルーチン本体プログラム

      RET
名前  ENDP
```

とします．このプロシージャを呼び出す（コール）には，

```
CALL   名前
```

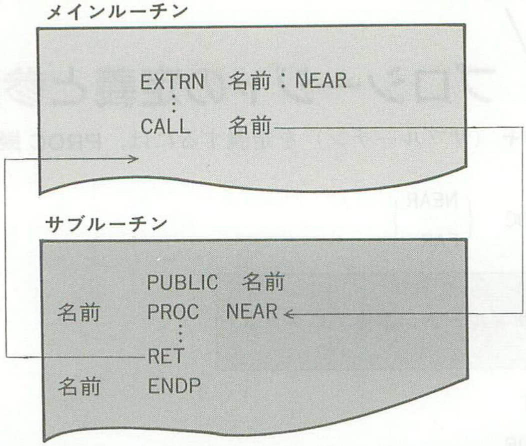
とします．名前のタイプであるNEAR，FARはプロシージャを定義するときに，そのどちらかに決められます．セグメント内のコールならばNEAR，セグメント外のコールならばFARを指定します．もし，指定がなければアセンブラはNEARと解釈します．

次に，CALL命令のあるメインルーチンとPROC擬似命令により定義されたプロシージャ（サブルーチン）が各々別のソース・ファイルにあり，別個にコンパイルされたとします．このときにメインルーチンの方ではプロシージャ（サブルーチン）の名前が定義されていないので，その名前は外部で定義して明示しておかなければなりません．

逆にプロシージャ（サブルーチン）側では‘名前’が外部から参照できるように明示しておかなければなりません．これらを行うのが，**EXTRN，PUBLIC擬似命令**です．

次のページの**図4.17**にメインルーチンからプロシージャ（サブルーチン）を参照する際のEXTRN，PUBLIC擬似命令の関係を示します．

図4.17 EXTRNとPUBLIC




## 4.12 モジュール別開発

長いプログラムを作成するときには，いくつかのモジュールに機能分割して，各モジュール毎にプログラムを作成し，適当に完成したところで各モジュールを一本にまとめて総合的なデバッグに入る開発方法をモジュール別開発と呼び，一般の高級言語ではよく用いられる手法です．

MASMはリロケータブルなオブジェクトを生成するので，このモジュール別開発という手法を採ることができます．

ここでは説明のために，各モジュールに極めて簡単なものを用いてモジュール別開発の概念を解説することにします．**図4.18**にメインモジュール，**図4.19**，**図4.20**にサブモジュールを示します．

▼図4.18 メインモジュール

```
A>TYPE B:MMODUL.ASM
; ----------------------------
; *      main module         *
; ----------------------------
;
PGROUP  GROUP   CODE
DGROUP  GROUP   DATA
```

```
DATA     SEGMENT WORD PUBLIC 'DATA'          ①
MSG      DB      0DH,0AH,'Hit key(1-9)!','$'
DATA     ENDS

CODE     SEGMENT BYTE PUBLIC 'CODE'
         ASSUME  CS:PGROUP,DS:DGROUP,SS:STACK
         EXTRN   KEYIN:NEAR,DISP:NEAR

START:   MOV     AX,DATA      ; DS set
         MOV     DS,AX
                                              ②
         MOV     AH,9
         MOV     DX,OFFSET MSG
         INT     21H
         CALL    KEYIN
         CALL    DISP
         MOV     AH,4CH       ; jmp MS-DOS
         INT     21H
CODE     ENDS

STACK    SEGMENT STACK
         DW      32 DUP(?)       ③
STACK    ENDS

         END     START
```

▼図4.19　サブモジュール１

```
A>TYPE B:SMODUL1.ASM
; -------------------------------
; *      sub module 1           *
; -------------------------------
;
PGROUP   GROUP   CODE
CODE     SEGMENT BYTE PUBLIC 'CODE'
         ASSUME  CS:PGROUP
         PUBLIC  KEYIN
KEYIN    PROC    NEAR
         MOV     AH,1                   ④
         INT     21H
         RET
KEYIN    ENDP
CODE     ENDS
         END
```

▼図4.20　サブモジュール２

```
A>TYPE B:SMODUL2.ASM
; ----------------------------
; *      sub module 2        *
; ----------------------------
;
PGROUP   GROUP   CODE
DGROUP   GROUP   DATA
CODE     SEGMENT BYTE PUBLIC 'CODE'
         ASSUME  CS:PGROUP,DS:DGROUP
         PUBLIC  DISP
```

```
DISP    PROC     NEAR
        MOV      CH,0      ; set count CX
        MOV      CL,AL
        SUB      CL,30H                          ⑤
START:  MOV      AH,9
        MOV      DX,OFFSET MSG
        INT      21H
        LOOP     START
        RET
DISP    ENDP
CODE    ENDS

DATA    SEGMENT  WORD PUBLIC 'DATA'
MSG     DB       0DH,0AH,'Hello MS-DOS !','$'    ⑥
DATA    ENDS

        END
```

メインモジュールは，‘Hit key（1～9）!’というメッセージを表示し，サブモジュール1（KEYIN）とサブモジュール2（DISP）を呼び出します（コールする）.

サブモジュールKEYINは，キーボードから入力したキーのアスキー・コードをレジスタALに与えてメインルーチンに戻ります（リターンする）.

サブモジュールDISPは，レジスタALに与えられた値－‘0’で示される回数だけ‘Hello MS-DOS！’というメッセージをCRTに表示します.

例では，各モジュールのCODE部はPGROUPに，DATA部はDGROUPにグループ化してありますが，必ずしもグループ化する必要はありません．PGROUPはBYTE PUBLIC，DGROUPはWORD PUBLICで結合します．したがって各モジュール間の参照はNEAR型となり，メインモジュールで，

```
EXTRN    KEYIN：NEAR,DISP：NEAR
```

と宣言し，

```
CALL   KEYIN
CALL   DISP
```

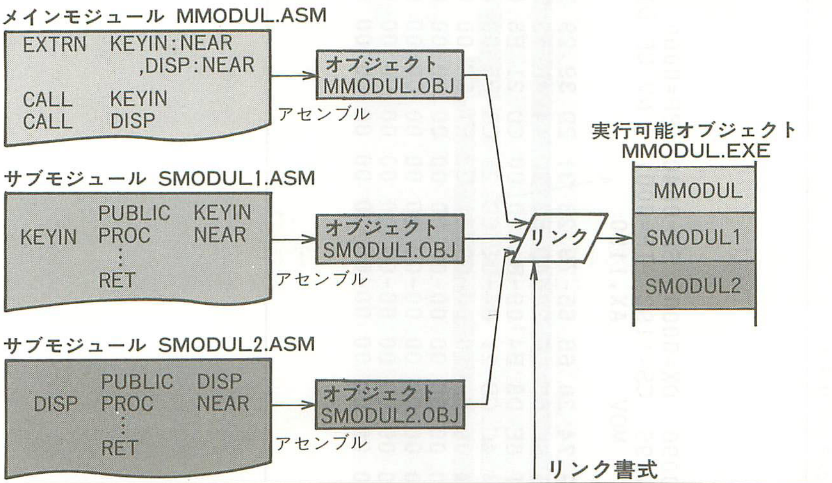
で外部モジュールを参照します.

参照されるモジュールはPROCを用いて，次のように行います.

```
          PUBLIC  KEYIN
KEYIN     PROC NEAR
            ：
          RET
KEYIN     ENDP
```

これら3つのモジュールのアセンブル／リンクの概要を**図 4.21** に示します．また，リンクされたロードモジュール MMODUL.EXE を DEBUG によりダンプしたものを**図 4.22** に示します．①～⑥の数字は**図 4.18**～**図 4.20** のソースプログラムの各セグメントに対応しています．

図4.21　各モジュールのアセンブル／リンクの概要




```
A>LINK ↵
Object Modules [.OBJ]: B:MMODUL+B:SMODUL1+B:SMODUL2 ↵
Run File [A:MMODUL.EXE]: B:MMODUL ↵
List File [NUL.MAP]: ↵
Libraries [.LIB]: ↵
```

▼図4.22　DEBUGによるMMODUL.EXEのダンプリスト

```
A>DEBUG B:MMODUL.EXE
-R
AX=0000  BX=0000  CX=0090  DX=0000  SP=0040  BP=0000  SI=0000  DI=0000
DS=1180  ES=1180  SS=1195  CS=1192  IP=0001   NV UP DI PL NZ NA PO NC
1192:0001 B89011        MOV     AX,1190
-DDS:100,18F
1180:0100  0D 0A 48 69 74 20 6B 65-79 28 31 2D 39 29 21 24    ..Hit key(1-9)!$   ①
1180:0110  0D 0A 48 65 6C 6C 6F 20-4D 53 2D 44 4F 53 20 21 ←  ..Hello MS-DOS !   ⑥
1180:0120  24 B8 90 11 8E D8 B4 09-BA 00 00 CD 21 E8 07 00 ←② $8...X4.:..M!h..
1180:0130  E8 09 00 B4 4C CD 21 B4-01 CD 21 C3 B5 00 8A C8 ←⑤ h..4LM!4.M!C5..H   ④
1180:0140  80 E9 30 B4 09 BA 10 00-CD 21 E2 F7 C3 00 00 00    .i04.:..M!bwC...
1180:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
1180:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ←③ ................
1180:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
1180:0180  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
```

# 4.13／条件アセンブル

**IF～ELSE～ENDIF 擬似命令**により条件を与えてアセンブルの流れを変更することができます．条件アセンブルの一般形は次のようになります．

```
IF 式
    ブロック1
ELSE
    ブロック2
ENDIF
```

もし，式が真（0以外）ならばブロック1を，偽（0）ならばブロック2をアセンブルします．ENDIF はブロックの終わりに必ず付けなければなりません．

条件アセンブルは，条件によりアセンブルの流れを制御するのであり，プログラム自体の実行の流れを制御するものではない点に注意してください．条件の書き方にはこの IF 以外にも何種類かあります．詳しい内容は**第5章5.14**を参考にしてください．

それでは，実際の条件アセンブルの例を124ページ**図4.23**に示します．

この例は MS-DOS というシンボルに与える値が真ならば‘Hello MS-DOS’のデータブロックが，偽ならば‘Hello CP/M-86’のデータブロックをアセンブルするというものです．

アセンブル結果を**図4.24**に示します．

▼図4.23 条件アセンブルの例

```
A>TYPE B:IF.ASM
; ---/*    if else endif example    */---
MSDOS     EQU       -1
          IF        MSDOS

DATA      SEGMENT
MSG       DB        'Hello MS-DOS'
DATA      ENDS

          ELSE

DATA      SEGMENT
MSG       DB        'Hello CP/M-86'
DATA      ENDS

          ENDIF

          END
```

▼図4.24 条件アセンブルの結果

```
                                        ; ---/*    if else endif example    */---
=-0001                                  MSDOS     EQU       -1
                                                  IF        MSDOS

0000                                    DATA      SEGMENT
0000      48 65 6C 6C 6F 20             MSG       DB        'Hello MS-DOS'
          4D 53 2D 44 4F 53
000C                                    DATA      ENDS

                                                  ENDIF

                                                  END
```

# 第5章

# MASM文法要約

MASM

## 本章の目的

前章でマクロアセンブラ(MASM)を使用する上での基礎的な概念について説明しましたが，ここでは各演算子および擬似命令の使い方について述べます．各演算子，擬似命令を機能別に分類し，各々について書式，機能，例について解説していきます．

# 5.1／MASMの演算子の種類

MASMで使用できる演算子には，次の表5.1に示すようなものがあります．

●表5.1　MASMの演算子

| | 演算子 | 機　　能 | 優先順位 | 参照 P. no. |
|---|---|---|---|---|
| オーバーライド | PTR | オーバーライドのタイプまたは距離を明示する | 3 | 128 |
| | : | セグメント・オーバーライド | 2 | 129 |
| | SHORT | ラベルまでの距離が±127バイトであることを示す | 11 | 130 |
| | THIS | タイプまたは距離を明示したオペランドを作る | 3 | 131 |
| | HIGH/LOW | 16ビット値の上位8ビット／下位8ビットを分離する | 4 | 132 |
| 値を返す | OFFSET | セグメント・ベースからの変数，ラベルの変位量を示す | 3 | 132 |
| | SEG | 変数，ラベルの置かれているセグメント・ベース値を返す | 3 | 133 |
| | TYPE | オペランドのタイプ，距離を返す | 3 | 134 |
| | .TYPE | 変数のモード，外部参照するか否かという特性を返す | 11 | 135 |
| | LENGTH | 変数のタイプの単位を返す | 1 | 135 |
| | SIZE | 変数に対して割り振られたバイト数の合計を返す | 1 | 136 |
| レコード指定 | シフト・カウント | レコードのLOW ENDからフィールドのLOW ENDまでのビット数 | 1 | 137 |
| | MASK | 指定したフィールドをすべて1，他をすべて0にマスクしたレコード値 | 1 | 137 |
| | WIDTH | フィールド，レコードに含まれるビット幅の数 | 1 | 138 |
| 算術 | ＊　/ | 乗除 | 5 | |
| | MOD | 剰余 | 5 | |
| | SHR /SHL | 右シフト／左シフト | 5 | 139 |
| | — | 負符号 | 6 | |
| | ＋　— | 加減 | 6 | |
| 関係 | EQ/NE | 等しい／等しくない | 7 | |
| | LT/LE | より小さい／より小さいか等しい | 7 | 139 |
| | GT/GE | より大きい／より大きいか等しい | 7 | |
| 論理 | NOT | 否定 | 8 | |
| | AND | 論理積 | 9 | |
| | OR | 論理和 | 10 | 140 |
| | XOR | 排他的論理和 | 10 | |
| マクロ | & | ‘’内や名前内の引数の置き換え | | 141 |
| | <> | <>内を単一のパラメータとして扱う | | 141 |
| | % | %に続く式を数値に変換する | | 142 |
| | ! | 次のキャラクタを文字の通りに扱う | | 143 |
| | ;; | ;;に続くコメントはマクロ展開されない | | 143 |

## 5.2 / オーバーライド演算子

変数およびラベルのセグメント，オフセット，タイプ，距離をオーバーライドするために用います．

### ● PTR（ポインタ）

【書式】

```
｛タイプ｝
｛距離　｝ PTR 式
```

タイプ…BYTE／WORD／DWORD
距離　…NEAR／FAR
式　　…変数／ラベル／メモリオペランド（[BX]，[SI] など）

【機能】

8086では，8ビット（バイト）単位と16ビット（ワード）単位の2種類のデータを扱うので，命令のオペランドにこれらのデータが使われたとき，そのデータのタイプを明示しなければならなくなります．

たとえば，BXレジスタの示すアドレス［BX］は型を持たないので，

```
MOV    [BX],1AH
```

としたのでは1AHがバイト単位なのかワード単位なのかはっきりしません．したがってアセンブラはエラー・メッセージを出します．そこで，もしバイト・データの転送ならば，

```
MOV    BYTE   PTR [BX],1AH
```

または，

```
MOV    [BX],BYTE   PTR 1AH
```

とします．なお，1AHを001AHと書けば，ワード・データであることがはっ

きりと示されます．

このように**PTR 演算子**はオペランドの型が不明だったり，異なっているときに一時的（この命令を行っている間だけ）にオペランドの型を変更するためによく用います．

また，JMP 命令や CALL 命令などで，セグメント内（NEAR）への分岐か，セグメント外（FAR）への分岐かを明示しなければならないときにも適用されます．

**【例】**

```
MOV   WORD   PTR    ARRAY,23 H   ←ARRAY 番地に 0023 H をワード転送する
MOV   BYTE   PTR [BX],23 H       ←［BX］番地に 23 H をバイト転送する
CALL   FAR   PTR   ABC           ←ラベル ABC へ FAR コールする
```

## ●：(セグメント・オーバーライド)

**【書式】**

｛セグメント・レジスタ／セグメント名／グループ名｝：アドレス式

セグメント・レジスタ…CS ／ DS ／ ES ／ SS
セグメント名　　　　…SEGMENT 擬似命令で定義された名前
グループ名　　　　　…GROUP 擬似命令で定義された名前
アドレス式　　　　　…変数／メモリオペランド（［BX］，［SI］など）

**【機能】**

アドレス式（メモリ・オペランド）を参照するには，それが置かれているセグメントのベースとそこからのオフセットがわからなければなりません．

一般には，ASSUME 擬似命令により参照が行われるセグメントは仮定されているので，各命令のオペランドにセグメントを指定する必要はありませんでした．しかし，ASSUME 擬似命令が使われていない場合や現在の ASSUME 命令で仮定されているセグメント以外を参照するには，**：演算子**によりセグメントオーバーライドしなければなりません．

【例】

MOV　AX,ES：［BX＋SI］　←ESレジスタが示すセグメント・ベースから，［BX＋SI］で示すオフセット値の分だけ離れたメモリの内容をレジスタAXに転送する

MOV　BX,OFFSET　DGROUP：ABC　←（DGROUP）グループ名のセグメント・ベースから測った変数ABCまでのオフセット値をレジスタBXに転送する

● SHORT

【書式】

SHORT　ラベル

【機能】

一般にラベルを前方参照する場合には，それへの距離がパス1のアセンブル時に不明なので，NEAR属性として2バイト・オペランドとして処理されます．もし，パス2で距離が±128バイトのSHORTの範囲であることがわかれば，オペランドには2バイトを必要としなくなるので，最後のバイトはNOP（90 H）に変えられます．たとえば，次のようなJMP命令は

```
JMP　ABC
```

| 10バイト離れているとする |
|---|

```
ABC：
　　⇩
EB　0B　90
```

という3バイトの命令になります．また，前方参照しているラベルが±128バイトの範囲にあれば**SHORT演算子**を用いて，

```
JMP　SHORT　ABC
```

とすることにより，EB 0Aという2バイトの命令で済みます．前者が0Bとい

うコードを生成しているのは，NOP(90 H)の分が余計に加わっているためです．

### ● THIS

【書式】

SHORT {距離／タイプ}

距離　…NEAR ／ FAR
タイプ…BYTE ／ WORD ／ DWORD

【機能】

**THIS 演算子**は，EQU または＝擬似命令と一緒に用いられることが多い演算子です．たとえば，

```
BARRAY   EQU    THIS   BYTE
ARRAY    DW     10   DUP ( ? )
```

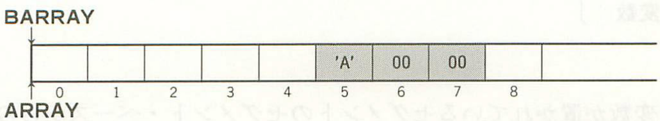
とすると，ARRAYがワード属性で定義されていても，この領域をBARRAYというバイト型変数を用いて参照することができるようになります．したがって，

```
MOV    BARRAY [ 5 ],‘A’
```

はバイト転送となり，

```
MOV    ARRAY [ 6 ],0000H
```

はワード転送となります．したがって次のようにデータは転送されることになります．




なお，THIS 演算子の代わりに LABEL 擬似命令を用いても同等な機能が行えます．

【例】

```
ABCF   EQU    THIS   FAR
ABC：←――――――――
```

：で定義されたABCというラベルはNEAR属性となるが，このポイントをABCFというFAR属性のラベルとしてもエントリできるようになる．

● HIGH／LOW

【書式】

HIGH／LOW　式

【機能】

HIGHは式（16ビット長）の上位8ビットを分離します．

LOWは式（16ビット長）の下位8ビットを分離します．

【例】

MOV　AL, HIGH　1A00H←レジスタALに1AHが転送される

## 5.3／値を返す演算子

● OFFSET

【書式】

OFFSET ｛ラベル／変数｝

【機能】

ラベルまたは変数が置かれているセグメントのセグメント・ベースからの変位量（オフセット）を返します．8086では，

```
MOV    DX,MSG
```

とすれば，レジスタ DX に MSG 番地の内容が転送されます．これに対して，

```
MOV    DX,OFFSET   MSG
```

という命令は，MSG が置かれているアドレスのオフセット値が転送されます．また，OFFSET 演算子を用いずに LEA 命令により，

```
LEA    DX,MSG
```

としても MSG のオフセット値が転送できます．

【例】

```
MOV    BX,OFFSET   DGROUP：ABC
```

…グループ化されたセグメントのグループ先頭からのオフセットを生成するには，グループ名 DGROUP によってオーバーライドしなければならない（詳細は第 4 章 4.8 を参照のこと）．

### ● SEG

【書式】

SEG ｛ラベル／変数｝

【機能】

ラベルまたは，変数が定義されているセグメントのセグメント値（セグメント・ベース・アドレス）を返します．

【例】

```
DATA   SEGMENT
    ⋮
ABC    DB
    ⋮
DATA   ENDS
```

```
    ⋮
MOV    AX,SEG   ABC
    ⋮              …これは，MOV    AX,DATA と等しい
```

## ● TYPE

**【書式】**

TYPE ｛ラベル／変数｝

**【機能】**

オペランドがラベルならば，次のような距離を示す値を返します．

NEAR（FFFFH）

FAR（FFFEH）

オペランドが変数なら次のようなタイプを示す値を返します．

BYTE（1）

WORD（2）

DWORD（4）

QWORD（8）

TBYTE（10）

STRUC（STRUCによって宣言されるバイト数）

**【例】**

```
ARRAY   DW   10   DUP（?）
            ⋮
        MOV  （TYPE   ARRAY）   PTR   ARRAY[5],0
                  …これは，MOV    WORD  PTR   ARRAY[5],0 と等しい
```

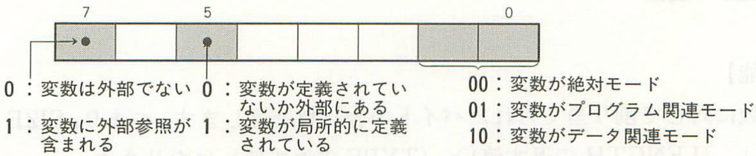
● .TYPE

【書式】

.TYPE　変数

【機能】

変数の特性を返します．特性に応じて次のビットがセットされます．




.TYPE 演算子はマクロ内部で条件アセンブルの条件として使われることが多い演算子です．

【例】

```
ABC     MACRO   DUMY
        LOCAL   P
P       =       .TYPE   DUMY
        IF   P
        ∫
```

…引数 DUMY へ渡された変数のタイプにより条件アセンブルを行う

● LENGTH

【書式】

LENGTH　変数

【機能】

変数に対して割り当てられたタイプの単位（BYTE，WORD，DWORD，QWORD，TBYTE）の数を返します．

【例】

```
ABC    DW   100   DUP（？）
MOV    AX，LENGTH   ABC…ABCの長さ100（WORD）が転送される
ABC    DD（？）…1タイプの単位しかないので，LENGTH　ABCは1を返す．
```

● SIZE

【書式】

SIZE　変数

【機能】

変数に対して割り当てられたバイトの合計数を返します．つまり，SIZEの返す値は，（LENGTHの返す値）×（TYPEの返す値）になります．

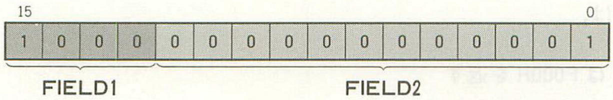
【例】

```
ABC    DW   100   DUP（？）
MOV    AX,SIZE   ABC  …（LENGTHの返す値）×（TYPEの返す値）＝（長さ100）×
                       （タイプ2バイト）＝200がレジスタAXに転送されます．
```

## 5.4 / レコード指定演算子

レコード指定演算子は，レコード内のフィールドを分離するのに使用します．レコードは，**RECORD擬似命令**で定義されます（本章**5.9**を参照のこと）．ここでは次のようにレコードが定義されているものとして説明を進めていきます．

```
TAG    RECORD   FIELD1：4,FIELD2：12
DATA   SEGMENT
ABC    TAG      ＜8,1＞
DATA   ENDS
```




ここで用いたTAGはレコードの型を示すタグ，ABCはTAGにより定義された変数，FIELD 1，FIELD 2はフィールド名をそれぞれ示します．

### ●シフト・カウンタ（レコードフィールド名）

【書式】

レコードフィールド名

【機能】

レコードフィールド名は，そのフィールドをレコードの右端まで右シフトするために必要なビット数の値を持っています．

先に上図で示した例では，

FIELD1は，12という値を持っている

FIELD2は，0という値を持っている

ということになります．

【例】

```
MOV    DX,ABC
MOV    CL,FIELD1
SHR    DX,CL          …ABCの内容が12ビット右にシフトするため，
                       8001 H → 0008 H
                      となり，FIELD 1の内容が右端にシフトされたことになる
```

### ● MASK

【書式】

MASK　レコードフィールド名

【機能】

レコードフィールド名で指定されるフィールドの各ビットに1を，それ以外には0をセットした値を返します．

先に示した例では，

MASK FIELD1 は F000H を返す

MASK FIELD2 は 0FFFH を返す

ということになります．

【例】

```
MOV    DX,ABC
AND    DX,MASK  FIELD1
       …          8001  ←となり，FIELD 1 以外を 0 にマスクしたことになる
         AND ) F000
                8000
```

● WIDTH

【書式】

WIDTH { レコードフィールド名 / レコード }

【機能】

レコードフィールドまたはレコードの幅をビット数で返します．

【例】

```
MOV    AL,WIDTH  FIELD2  …レジスタ AL に 12 が転送される
MOV    CL,WIDTH  TAG  …レジスタ CL に 16 が転送される
```

# 5.5／算術／関係／論理演算子

## ●算術演算子

算術演算子は，**表 5．2** に示すように 8 種類の演算子があります．＋および－演算子には，定数以外のオペランドが書けますが，それ以外の算術演算子ではオペランドとして定数しか書くことができません．

**定数**とは，10，0011 B，1 AH などの数値定数や EQU などで定義された名前（シンボル）を指します．

●表5.2　算術演算子

| 演算子 | 機　能 | 例 |
|---|---|---|
| ＊ | 乗算 | 10＊20，ARRAY＊2 |
| ／ | 除算 | 20／5，ARRAY／5 |
| MOD | 剰余 | 10 MOD 3 → 1 |
| SHR | 右シフト．空いた左ビットには 0 が入る | 11001B SHR 2<br>→2 ビットシフトされて 110B となる |
| SHL | 左シフト．空いた右ビットには 0 が入る | 11001B SHL 2 →1100100B |
| － | 負符号 | －100 |
| ＋ | 加算 | ARRAY＋10，[BX]＋[SI] |
| － | 減算 | ARRAY－10，[BX]－5 |

## ●関係演算子

関係演算子は，2 つの定数オペランドの大小を比較し，2 つのオペランドの関係が演算子と一致したならば，真（FFFFH：－1）を返し，一致しなければ偽（0）を返します．

関係演算子は，条件擬似命令や条件命令とともによく使用されます．たとえば次のように使われます．

```
ABC    EQU    10
XYZ    EQU    20
        ⋮
```

```
IF      ABC    LT   XYZ
   文 1
ELSE
   文 2
ENDIF
```

●表5.3 関係演算子

| 演算子 | 機　能 |
|---|---|
| EQ | 等しい |
| NE | 等しくない |
| LT | より小さい |
| LE | より小さいか等しい |
| GT | より大きい |
| GE | より大きいか等しい |

### ●論理演算子

論理演算子は，２つの定数オペランドのビット毎の演算を行います．関係演算子などと組み合わせて，真／偽の論理演算を行いたい場合は，すべての定数オペランドにFFFFH（－１：真）および０（偽）を用いなければなりません．

たとえば，真／偽の論理演算をするならば，

```
IF  (ABC   LT  XYZ) AND  (ABC  EQ   100)
```

などのようにします．ビットごとの論理演算をする場合は，

```
MOV    AX,0F0FH   AND   00FFH
```

などとします．

●表5.4 論理演算子

| 演算子 | 例 |
|---|---|
| NOT | NOT 0000H → FFFFH |
| AND | 0A1AH AND 00FFH → 001AH |
| OR | 0A1A　OR　00FFH → 0AFFH |
| XOR | 0F0FH XOR FFFFH → F0F0H |

# 5.6 ／マクロ演算子

マクロブロック内で使用される特殊なマクロ演算子は次の５種類があります．

### ●＆（アンパサンド）

マクロ本体のダミーが名前や単一引用符（シングル・クォーティションマーク）の中に入っているときに，それがマクロのダミーであることを明示するための演算子です．

たとえば，

```
FILL    MACRO Y
LP&Y：  MOV    AL,'&Y'
        ENDM
```

というマクロを，

```
FILL    Z
```

としてマクロコールすると，

```
LPZ：   MOV   AL,'Z'
```

として展開されます．もし，＆を付けずに，LPY や'Y'としたならば，ダミーY の置き換えは行われません．

### ●<＞

マクロコール時のパラメータは，コンマ（,）または空白によって区切られるため，

```
MSG    MACRO   DUMY
       DB      '&DUMY'
       ENDM
```

というマクロを，

```
MSG    good   night
```

のように呼び出すと，パラメータにgoodとnightの2つがあるとみなして，DUMYにはgoodだけを渡します．good night を1つのパラメータとしてDUMMYに引き渡すには，

```
MSG   <good   night>
```

とします．＜と＞に挟まれた内容が単一のパラメータとしてDUMYに引き渡されることになります．

### ●%

%に続くシンボルを文字列としてではなく，そのシンボルに定義されている値をマクロに引き渡します．

たとえば，

```
MSG    MACRO   DUMY
       DB     ' &DUMY'
       ENDM
```

というマクロを，

```
ABC    =      1
      MSG    %ABC
```

のように呼び出すと，

```
DB    '1'
```

とABCの値が渡されますが，これを，

```
MSG   ABC
```

のように呼び出すと，

```
DB    'ABC'
```

と普通に展開されます．

●！

！に続くキャラクタを文字通りに（演算子や擬似命令などと解釈しないで）アセンブラに認識させます．たとえば'；ABC'などの文字列をマクロに渡したいときは，単に次のように，

```
MSG   ；ABC
```

としたのでは，；はコメントとして認識されます．そこで次のように，

```
MSG   ！；ABC
```

とします．結果として，

```
MSG   <；ABC>
```

と同じことになります．

●；；

.LALL 擬似命令（ディフォルトは .XALL）が指定されているとコメントを含むすべての命令をマクロ展開することになります．そのときに，コメントに二重の；；を付けておくと，そのコメントはマクロ展開されません．

# 5.7 MASMの擬似命令の種類

MASMで使用できる擬似命令を表**5.5**に示します

●5.5 MASMの擬似命令

| | 擬似命令 | 機能 | 参照P. |
|---|---|---|---|
| セグメント | SEGMENT～ENDS | セグメントの定義 | 145 |
| | GROUP | いくつかのセグメントを単一の名前で参照できるようにグループ化する | 148 |
| | ASSUME | どのセグメント・レジスタを用いてセグメントを参照するかを仮定する | 149 |
| データ定義 | DB/DW/DD/DQ/DT | バイト／ワード／ダブルワード／クォドワード／テンバイト単位でデータを定義（Define）する | 150 |
| | STRUC～ENDS | 構造体型データの型定義 | 152 |
| | RECORD | ビットフィールドで区切ったレコード型の定義 | 155 |
| プロシージャ・モジュール | PROC～ENDP | プロシージャ（手続き）の定義 | 156 |
| | EXTRN | 他のモジュールで定義されている名前であるという宣言 | 157 |
| | PUBLIC | 他のモジュールから参照される名前であるという宣言 | 159 |
| | NAME | モジュールに名前を付ける | 160 |
| マクロ | MACRO～ENDM | マクロブロックの定義 | 165 |
| | LOCAL | マクロ内での局所的な名前であるという宣言 | 166 |
| | EXITM | マクロからの抜け出し | 167 |
| | PURGE | マクロ定義の削除 | 168 |
| 繰り返し | REPT | 繰り返し | 169 |
| | IRP | 不定回数の繰り返し | 169 |
| | IRPC | 不定回数の文字繰り返し | 169 |
| 文の制御 | END | プログラムの終わりと開始番地の指定 | 160 |
| | EQU | 式の値を名前に割り当てる | 161 |
| | = | 式の値を名前に割り当てる，EQUと違い再定義可能 | 162 |
| | LABEL | 名前にタイプを定義する | 162 |
| | .RADIX | 進数（基数）の変更 | 163 |
| | ORG | ロケーション・カウンタを式の値にセットする | 163 |
| | EVEN | プログラム・カウンタを偶数番地に進める | 164 |
| | COMMENT | 任意の行に渡ってコメントを書く | 164 |
| | INCLUDE | 現ソース・ファイルに他のファイルを取り込む | 165 |
| 繰り返し | IFxxxx～ELSE～ENDIF | 条件が真ならIFブロックを，偽ならばELSEブロックをアセンブルする．xxxxの種類によって各種条件判断できる | 171 |
| | IF/IFE | 式が0でない／0であるかを判断する | 171 |
| | IF1/IF2 | アセンブラがパス1／パス2のどちらを実行中か判断する | 171 |

| | | | |
|---|---|---|---|
| | IFDEF/IFNDEF | シンボルが定義されている／いないかを判断する | 172 |
| | IFB/IFNB | 引数がブランクである／でないかを判断する | 172 |
| | IFIDN/IFDIF | 2つの引数文字列が同じ／同じでないかを判断する | 173 |
| リスティング | PAGE | ページサイズの指定およびページの開始 | 174 |
| | TITLE/SUBTTL | タイトル／サブタイトルの指定 | 174 |
| | %OUT | アセンブル中にメッセージをCRTに表示する | 175 |
| | .LIST/.XLIST | すべての行をリストする／しない | 175 |
| | .LFCOND/.SFCOND | 偽条件ブロックをリストする／しない | 176 |
| | .TFCOND | ／Xで指定した偽条件ブロックリストモードを反転する | 176 |
| | .XALL/.SALL | マクロによって作成したコードをリストする／しない | 177 |
| | .LALL | コメント行も含めてマクロ展開し，すべてをリストする | 177 |
| | .CREF/.XCREF | クロスリファレンスの作成を開始／抑止する | 178 |

## 5.8／セグメント擬似命令

MASMではセグメントの概念が重要です．このセグメントを定義したり，セグメントとセグメント・レジスタの関係を仮定する擬似命令をこれから説明します．

### ● SEGMENT〜ENDS

【書式】

セグメント名　SEGMENT　［アラインメント］［コンバインタイプ］［'クラス名'］
　　　　　　　⋮
セグメント名　ENDS

【機能】

コードとデータを生成するすべての命令は，いずれかのセグメント内に置かなければなりません．セグメントは大別すると，コード（命令），データ，スタックの3種類になります．

このセグメントに名前を付けたり，セグメント開始境界（アラインメント）の設定，セグメントの組み合わせ（コンバインタイプ）などを指定するのがSEGMENT擬似命令です．

・セグメント名

　セグメント名は名前としての規則を守っていれば，何であっても構いませ

ん．同一ソース・プログラムの中に同じセグメント名を持つセグメントは，アセンブルする際に，ひとつの論理セグメントとしてまとめられます．したがって，同一ソース・プログラム中にある同じ名前のセグメントのアラインメント，コンバインタイプ，クラス名は同じでなければなりません．

・アラインメント

セグメントの開始境界を次の4つの方法で指定できます．

BYTE（バイト境界） …データが前のセグメントの直後から隙間なく置かれます．

WORD（ワード境界） …偶数アドレスをセグメント先頭にします．したがって前のセグメントとの間に最大で1バイトの空きが生じます．8086では，データがワード境界におかれている場合に最も効率よく参照できます．

PARA（パラグラフ境界） …16の倍数アドレス（xxxx 0 H）をセグメント先頭にします．アラインメントの指定がなければ，アセンブラはPARAとみなします．

PAGE（ページ境界） …256の倍数アドレス（xxx 00 H）をセグメント先頭にします．

・コンバインタイプ

‘クラス名’を持つセグメントを結合するときの組み合わせタイプをリンカに告げるためのもので，次の6タイプがあります．

無指定（PRIVATE） …個々のセグメントは論理的に独立しているため，各々独自のセグメント・ベースを持ちます．

PUBLIC …同じ‘クラス名’またはセグメント名のセグメントが隣接してロードされて単一の論理セグメントとなります．セグメント・ベースは最初のセグメントの先頭です．

STACK …PUBLICと同じ機能ですが，スタック・ポインタSPをスタック・セグメントの終端（スタック・ボトム）に初期設定する動作が余分に含まれています．

COMMON …同じ‘クラス名’のセグメントが互いにオーバーラップしてロードされます．

MEMORY …指定したセグメントを他の論理セグメントよりも上位に置きます．ただし，MS-LINKはこの機能をサポートしていません．

AT ＜式＞ …セグメントを＜式＞で指定したパラグラフ・アドレスに置きます．

・クラス名

クラス名は引用符（'）で囲まなければなりません．MS-LINKにより同じクラス名のセグメントは隣接してロードされます．リンカはセグメント名よりも，クラス名を優先してロードします．

【例】

```
CODE  SEGMENT  BYTE  PUBLIC  'CODE'  ┐
      ⋮                              │ コードセグメント
CODE  ENDS                           ┘

DATA  SEGMENT  WORD  PUBLIC  'DATA'  ┐
      ⋮                              │ データセグメント
DATA  ENDS                           ┘

STACK  SEGMENT  STACK                ┐
      ⋮                              │ スタックセグメント
STACK  ENDS                          ┘
```

## ● GROUP

【書式】

グループ名　GROUP　セグメント名［，…］

【機能】

GROUP擬似命令は，指定した異なるセグメント名のセグメントに単一のグループ名を割り当てます．このようにGROUP宣言した後，グループ名にASSUME擬似命令を用いることで，異なるセグメントに対して，CS，DSの値を変えることなく参照できるようになります．

つまり同一のグループ名で定義された異なるセグメントは，すべて単一のセグメント・ベース（最初のセグメントの先頭）を持つことになります．しかし，**OFFSET 演算子**により変数のオフセット値を得る場合は，その変数が置かれているセグメントの先頭からになります．もし，グループ先頭からのオフセット（実際にはこのような要求が多い）を得る場合は，グループ名を用いて次のようにオーバーライドしなければなりません．

```
OFFSET グループ名：変数名
```

なお，関連事項が第 4 章 **4.8** にあるので参考にしてください．

**GROUP 擬似命令**によるグループのサイズは，合計で 64 KB 以下でなければなりません．また，特定のコンパイラ（C，Pascal，Fortran）では，規定のグループ名，クラス名を用いてコード・グループ，データ・グループにグループ化するので，これらとリンクする場合にはそのコンパイラの規定に従わなければなりません．

**【例】**

```
DATA1   SEGMENT
MSG1    DB   'hello'
DATA1   ENDS

DATA2   SEGMENT
MSG2    DB   'MS-DOS'
DATA2   ENDS

CODE    SEGMENT
DGROUP GROUP   DATA1,DATA2 ←── DATA1，DATA2 セグメントに DGROUP
                               というグループ名を付ける
        ASSUME  CS：CODE,DS：DGROUP ←── DGROUP は DS レジスタを
                                        用いて参照すると仮定
        MOV     AX,DGROUP
        MOV     DS,AX
           ：
        MOV     AL,MSG2
```

```
MOV    BX, OFFSET  DGROUP：MSG2
```

## ● ASSUME

【書式】

ASSUME ｛セグメント・レジスタ：セグメント名［，…］
　　　　　NOTHING ｝

【機能】

**ASSUME 擬似命令**は，セグメントまたはグループ内の名前を，指定（仮定）したセグメント・レジスタで参照できることをアセンブラに指示します．

この命令では，セグメント・レジスタにセグメントの先頭アドレスをセットするのではない，ということに注意してください．

1つのセグメント・レジスタで複数のセグメントに対して，同時にASSUME擬似命令を行うことはできません．したがって，

```
ASSUME  DS：DATA 1
   ⋮
ASSUME  DS：DATA2
```

とすると，これ以後はDSを用いてDATA 1にはアクセスすることはできません．また，ASSUME　NOTHINGとすると，それまでのASSUME擬似命令で設定したものはすべてキャンセルされます．

・セグメント・レジスタ

ASSUME擬似命令で指定できる＜セグメント・レジスタ＞は，CS，DS，ES，SS です．

・セグメント名

指定できる＜セグメント名＞は，次のようなものです．

1. SEGMENT擬似命令で宣言された名前
2. GROUP擬似命令で宣言された名前
3. SEG演算子を用いた式

**【例】**

```
ASSUME   CS : CODE,DS : DATA,SS : STACK
```

…CODEセグメントはCSを，DATAセグメントはDSを，STACKセグメントはSSを介して参照できるということをアセンブラに指示する

# 5.9 データ定義擬似命令

単純な型のデータはDB, DW, DD, DQ, DTなどの**DEFINE擬似命令**を用いて定義でき，さらに複数（配列的）のデータはDEFINE擬似命令に**DUP**（**duplicate：複写**）を組み合わせて定義することができます．

CやPascalの構造体（レコード）型のデータに似たデータ型は**STRUC擬似命令**，ビットフィールドの定義は**RECORD擬似命令**によって行えます．

## ● DB／DW／DD／DQ／DT

**【書式】**

```
          ⎧ DB ⎫
          ⎪ DW ⎪
[変数名]  ⎨ DD ⎬  式 [, …]
          ⎪ DQ ⎪
          ⎩ DT ⎭
```

**【機能】**

DEFINE擬似命令が置かれているロケーションから，式で示されるデータをDの後に続く文字（B，W，D，Q，T）によって決まる単位で初期設定します．

変数名が指定されていれば，そのロケーションに変数名が（B，W，D，Q，T）によって決まる属性で定義されます．つまりDBで定義された変数名はバイトオペレーションとして使用できます．ただし，変数名の後にコロン（：）を付けると，変数でなくラベルになってしまうので注意してください．

DB（Define Byte）…1バイト

DW（Define Word）…1ワード

DD（Define Double Word）…2ワード

DQ（Define Quad Word）…4ワード

DT（Define Ten Bytes）…10バイト

式はコンマで区切ることで複数の式を書くことができます．式としては次のようなものを書くことができます．

1．定数式　…10や1AH＊RECなど

2．？　…初期設定をしないで領域の確保だけを行う

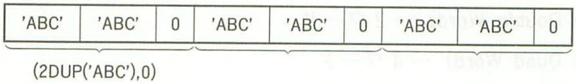
3．アドレス式　…ARRAYなどの変数名を書くとそのアドレス値が定義される．アドレス式の指定はDWとDDのみ指定できる

4．アスキー文字列…単一引用符（'）で囲まれる文字列．DBの場合だけは3文字以上の長さの文字列を書くことができる

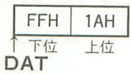
5．式DUP（式）…（　）内の式（コンマで区切って複数の式を書いてもよい）で指定される値を，先頭の式で指定された回数だけ繰り返し定義する．（　）内の式に？を用いると初期設定を行わない．また，DUPを入れ子（ネスト）にして書くこともできる

【例】

```
CR       EQU  0 DH
          ⋮
ARRAY    DB   'hello', CR, 0 AH…CR は 0 DH として定義されている
BUFF     DB   10    DUP(?)…10 バイト定義されるが，初期化はされない
MSG      DB   3     DUP(2  DUP('ABC'), 0)…次の図のように定義される
```




```
DAT      DW   1 AFFH…次の図のように下位，上位バイトの順に格納される
```




```
ADR      DW   ARRAY   …ARRAY という変数名のオフセット値が定義される．OFF-
                        SET 演算子は用いなくてもよい
ABC      DW   ?, ?…2 ワード定義されるが初期設定されない
XYZ      DW   'AB'…'B'，'A'の順に格納される
ADR      DD   ARRAY…ARRAY のオフセット，ARRAY のセグメント値の順に定義される
MAX      DD   4294967295…4 バイト最大値 FFFFFFFF が定義される
```

## ● STRUC～ENDS

【書式】

```
構造体タグ名   STRUC
メンバ名       DEFINE 擬似命令    式 [, …]
     ⋮
     ⋮
構造体タグ名   ENDS
```

【機能】

STRUC～ENDS 擬似命令は，DEFINE 擬似命令により定義される複数のデ

ータ群（タイプは異なっていても構わない）に総括的な名前（これを**構造体タグ名**と呼ぶ）を定義します．個々のDEFINE擬似命令で定義されたデータに付けられた変数名を特に**メンバ名**と呼びます．

STRUCは単に構造体タグ名の型を宣言するだけの擬似命令で，データ領域へ具体的なデータの定義を行うわけではありません．したがってSTRUC宣言はセグメントの外に置いても構いません．

データ・セグメント内に，構造体タグ名で示される型のデータを定義するには次のようにします．

```
DATA   SEGMENT
          ⋮
構造体変数名   構造体タグ名  <初期値>
          ⋮
DATA   ENDS
```

このデータの各要素（メンバ）を参照する場合は，ピリオド（.）を用いて次のように行います．

```
構造体変数名 . メンバ名
```

STRUC～ENDS内でDEFINE擬似命令で定義した各メンバの値はディフォルト値です．データ・セグメント内に構造体データを定義するときに，<>による初期化を行わなければこの値が定義されます．

なお，STRUC～ENDS内のメンバが複数のフィールドで定義されていたり，DUPを用いて定義されているときには，そのメンバを<>で初期化することはできません．初期化できないメンバは<>内で，コンマ（,）によりスキップします．たとえば，

```
TAG   STRUC
A     DB     'AAAAA'             ←初期化できる
B     DB     1, 2               ｝
C     DB     10 DUP ('A')       ｝初期化できない
D     DW      ?                  ←初期化できる
```

```
TAG   ENDS
```

という構造体で，構造体変数名 ABC をデータ・セグメント内に定義するには，

```
ABC   TAG  <'Candy',,, 80>
                   ↑↑
                   └──メンバB，Cは初期化できないのでスキップする
```

などとします．もし初期化を行わない場合は，

```
ABC   TAG  <>
```

とします．＜＞は省略することはできません．

STRUC 内で定義したデータのサイズと＜＞による初期化の際のデータのサイズが異なるとエラーとなりますが，文字列の場合だけはエラーとはならず，＜＞内の長さが長ければ，余分な文字列は無視されます．また，＜＞内の長さが短ければ，その分の空白は埋められます．

STRUC 内に書くことできる命令は，DEFINE 擬似命令とコメントだけです．なお，構造体変数ABCのメンバDの内容をレジスタAXに転送するには，

```
MOV   AX, ABC.D
```

とするか，または，

```
MOV   BX, OFFSET   ABC
MOV   AX, [BX].D
```

とします．

## ● RECORD

【書式】

レコード名　RECORD　フィールド名：幅［＝初期値］［，…］

【機能】

RECORD 擬似命令は，指定したビット幅で，レコード（8 ビット長または 16 ビット長）をフィールドに区切り，このフィールドおよびレコードに名前を定

義します．初期値を指定するとそのフィールドが初期設定されます．

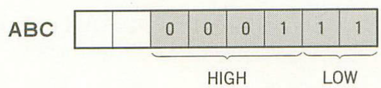
STRUC と同様に **RECORD 擬似命令**は，レコードの型を宣言するだけです．データ・セグメント内へ具体的なデータを定義するには次のように，

```
変数名 レコード名 <初期値>
```

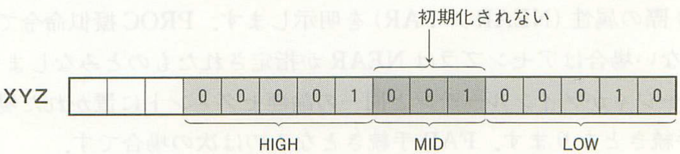
とします．

RECORD 擬似命令で定義されるフィールドの合計が 8 ビット以下の場合は 8 ビット・レコード，8 ビットを超える場合は 16 ビット・レコードとなり，レコード内のフィールド位置は一番最後に指定されたフィールドがレコードの右側（低位ビット）に取られます．

```
REC 1   RECORD   HIGH : 4, LOW : 2
    ⋮
ABC     REC 1    < 1,   3 >
                   ↑    ↑
                 HIGH  LOW
```




```
REC 2   RECORD   HIGH : 5, MID : 3=1, LOW : 5
    ⋮
XYZ     REC 2    < 1, , 2 >
        HIGH ————↑ ↑  ↑—— LOW
                  MID
```




また，REC 2 型のレコードを複数個定義するには，DUP を用いて

```
XYZ     REC2     10 DUP (<1,, 2>)
```

とします．なお，レコードの使用例は本章 **5.4** を参考にしてください．

# 5.10 / プロシージャ/モジュール擬似命令

プロシージャを定義する**PROC〜ENDP 擬似命令**, 外部モジュール間の名前を参照するための**EXTRN, PUBLIC 擬似命令**, そしてモジュールに名前を付ける**NAME 擬似命令**について説明します.

## ● PROC

【書式】

```
手続き名  PROC   ┌NEAR┐
                 └FAR ┘
            ⋮
          RET
手続き名  ENDP
```

【機能】

重複する処理をひとまとめにして, メインルーチンから必要に応じて呼び出すようにしたものを**サブルーチン**といいますが, これに名前を付けて, より構造的に表現できるようにしたものを**プロシージャ(手続き)**と呼びます.

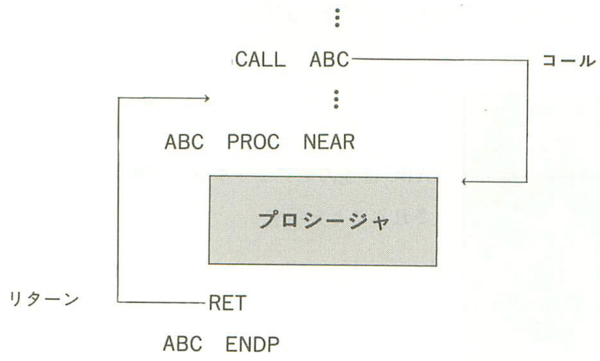
**PROC 擬似命令**はサブルーチンに手続き名と, その手続きが呼び出される(CALL)際の属性(NEAR/FAR)を明示します. PROC 擬似命令で属性の指定をしない場合はアセンブラは NEAR が指定されたものとみなします.

プロシージャがメインルーチンと同一の論理セグメントに置かれた場合は, NEAR 手続きとなります. FAR 手続きとなるのは次の場合です.

・手続き名が OS のエントリ名である場合
・手続きが他の論理セグメント(異なる ASSUME CS)から呼び出される場合

PROC〜ENDP には RET 命令が必ず置かれなければなりません. RET 命令の属性にも, PROC 擬似命令で定義した NEAR/FAR の属性が適用されます.

【例】




● EXTRN

【書式】

```
EXTRN　名前：タイプ［，…］
```

【機能】

**EXTRN 擬似命令**は，使用される名前がその**モジュール外**（external）の他のモジュールで定義されているということを宣言します．なお定義されている外部のモジュール内では，その名前が**PUBLIC宣言**されていなければなりません．

タイプには，名前（ラベル，変数，ナンバ）に対応した次のタイプが指定できます．

1. NEAR ／ FAR
2. BYTE ／ WORD ／ DWORD
3. ABS

アセンブラは EXTRN 宣言されている名前がそのセグメント内にあるものと仮定します．もしも異なる論理セグメント内の名前を参照するならば，EXTRN 宣言をセグメントの外に置き，

```
ASSUME セグメント・レジスタ：SEG　名前
```

として **ASSUME 擬似命令**を用いる必要があります．

【例】

・異なるモジュールが同一セグメントにロードされる場合

**モジュール1**

```
CODE   SEGMENT
       PUBLIC   ABC      ← ABCは他のモジュールから参照されることを示す
ABC    PROC     NEAR
          ⋮
       RET
ABC    ENDP
CODE   ENDS
```

**モジュール2**

```
CODE   SEGMENT
       EXTRN   ABC：NEAR  ← ABCは外部モジュールであり，NEAR属性で定義されていることを示す
       ASSUME  CS：CODE
          ⋮
       CALL    ABC
          ⋮
CODE   ENDS
```

・異なるモジュールが異なるセグメントにロードされる場合

**モジュール1**

```
DATA   SEGMENT
       PUBLIC   XYZ
XYZ    DB       'Hello'
          ⋮
DATA   ENDS
```

**モジュール 2**

```
        EXTRN   XYZ : BYTE
CODE    SEGMENT
        ASSUME  CS : CODE, DS : SEG XYZ
        MOV     AX, SEG  XYZ
        MOV     DS, AX
          ⋮
        MOV     AL, XYZ
          ⋮
CODE    ENDS
```

EXTRN XYZ : BYTE ← これをセグメント内で定義すると，XYZへの参照はCS：でオーバーライドされる

SEG XYZ ← 変数XYZが定義されているセグメント値を与える．DATAは外部モジュールのセグメント名であるため，ここにDATAとは書けない

## ● PUBLIC

【書式】

PUBLIC 名前 [, …]

【機能】

**PUBLIC 擬似命令**は，使用される名前が，当該モジュールで定義され，かつ他のモジュールからも参照されるということを宣言します．

名前には，ナンバ，変数，ラベルを指定できますが，2バイトを超える整数や小数をEQUで定義したシンボルは指定できません．

## ● NAME

【書式】

NAME モジュール名

【機能】

**NAME 擬似命令**は，モジュールに名前を付けます．モジュール名は何文字でも構いませんが，先頭の6文字だけが有効になります．MASMは次の順序でモジュール名を設定します．

1．NAME 擬似命令によって定義された名前
2．NAME 擬似命令がない場合は TITLE 擬似命令で定義された名前
3．NAME，TITLE 擬似命令がない場合は MASM はモジュール名を‘A’とする

## 5.11 / 文の制御に関する擬似命令

MASM の『マクロアセンブラマニュアル』では，本章 **5.8〜5.10** に示した擬似命令とこれから説明する擬似命令を一括して**メモリ擬似命令**と呼んでいますが，本書ではこれをもう少していねいに分類することにします．本章 5.8〜5.10 に示したもの以外の擬似命令を，ここでは**文の制御に関する擬似命令**として説明することにします．

### ● END

【書式】

END［式］

【機能】

**END 擬似命令**は，アセンブラにプログラムの終わりを指示します．

式が指定されると，それがプログラムの開始アドレスとなります．いくつかのモジュールを結合する場合に，メインモジュールのみに「END 式」が指定できます．式が書かれなかった場合は，リンカがリンクしたときの先頭モジュールの先頭をプログラムの開始番地として設定します．

【例】

```
START：                  プログラムがSTART番地から実行されるように,
        ：               CS：IPをセットする
        END    START
```

## ● EQU（Equate）

【書式】

名前　EQU　式

【機能】

**EQU 擬似命令**は，使用する名前を式の値に割り当てます．この結果として，プログラム中の名前はすべて，式の値に置き換えられます．これは単純なマクロ置換と考えられます．

一度EQUにより名前を定義すると，再定義することはできません．再定義したい場合は，＝（イコール）擬似命令を用いて行います．

EQU擬似命令に使うことのできる式には次のものがあります．

1. シンボル …既に定義されている名前
2. 命令 …MOVなどのオペコード
3. 有効な式 …10，20＊5，ADR＋5など
4. テキスト …［BP＋8］，6.3E7など

＊MASM ver 1.20ではこのテキストをEQUしたものをオペランドにするとエラーとなるバグがある

【例】

```
= 0200     LDはMOVと定義されて      SEC     EQU      512     ┐←有効な式
= 0400     いるので正しくアセンブ    REC     EQU      SEC*2   ┘
=          ルされた                  LD      EQU      MOV     ←命令
=                                    ABC     EQU      START   ←シンボル
0000                                 CODE    SEGMENT
                                             ASSUME   CS:CODE
0000  B8 0200                        START:  LD       AX,SEC
0003  BB 0400                                LD       BX,REC
0006  EB F8                                  JMP      ABC
0008                                 CODE    ENDS                ↖
                                                                 STARTに定義されている
                                             END
```

## ●＝（イコール）

【書式】

名前 ＝ 式

【機能】

**＝擬似命令**は，使用する名前を式の値に割り当てます．EQUと同じ機能ですが，次の点が異なります．

・式としては〈有効な式〉しか書けない
・再定義可能である

【例】

```
SEC  ＝  512
SEC  ＝  SEC＋ 1
```

## ● LABEL

【書式】

名前 LABEL タイプ

【機能】

**LABEL 擬似命令**は，直後に定義されている名前に異なるタイプの名前を割り当てます．

名前がラベルであるか変数であるかによって，タイプはNEAR／FARまたはBYTE／WORD／DWORD／ストラクチャ名／レコード名が指定できます．

**THIS 演算子**と**EQU**または**＝擬似命令**を用いてもLABEL擬似命令と同様な機能が得られます．本章**5.2**を参照してください．

【例】

```
BARRY  LABEL  BYTE
ARRAY  DW     10  DUP（？）
```

…ARRAYはワード属性で定義されているが，この領域のデータをBARAYとい

うバイト属性の変数を用いて参照することもできる

```
ABCF   LABEL   FAR
ABC：
```

…：で定義されたABCというラベルは，NEAR属性となるが，このポイントをABCFというFAR属性のラベルとしてもエントリできるようになる

## ●.RADIX

【書式】

```
.RADIX  式
```

【機能】

ディフォルトされている進数は，10進数ですが，**.RADIX擬似命令**によりディフォルトの進数を式で示される進数に変更します．

式には2から16の値を設定し，その値は現在のディフォルトされている進数に関係なく10進で表します．

.RADIX擬似命令は，DD／DQ／DT擬似命令には影響を与えません．

【例】

```
.RADIX   16
MOV      BL,10      … 16進の10Hとみなされる
MOV      BL,10D     … 10進の10とみなされる
```

## ●ORG（Origin）

【書式】

```
ORG  式
```

【機能】

**ORG擬似命令**は，ロケーション・カウンタが式の値にセットされ，アセンブラはその値で始まる番地からコード（データ）を割り当てます．

**$記号**は現ロケーション・カウンタの値を持ちます．

【例】

```
ORG   100 H          … ロケーション・カウンタを 100 H にする
ORG   $+2            … ロケーション・カウンタを 2 だけ進める
```

### ● EVEN

【書式】

```
EVEN
```

【機能】

**EVEN 擬似命令**は，プログラム・カウンタを偶数番地に進めます．このために必要に応じて NOP 命令が挿入されます．

BYTE アラインを指定したセグメントで EVEN を行うとエラーになります．

### ● COMMENT

【書式】

```
COMMENT  区切り記号
   ⋮
         区切り記号
```

【機能】

**COMMENT 擬似命令**の後に置かれる区切り記号（空白以外の文字）から，次にこの区切り記号が現れるまでの複数行をコメント文とみなします．

；による1行のコメント文でなく，複数行のコメントが記述できます．

【例】

```
COMMENT  *  ←区切り記号
  This program is written by…  ┐
     ⋮                          ├ コメント文
         *                      ┘
```

### ● INCLUDE

【書式】

```
INCLUDE  [d:]filename.typ
```

【機能】

**INCLUDE 擬似命令**が置かれている位置に，filename.typ のソース・ファイルを取り込みます．取り込まれた行のリスティング・ファイルには，‘C’という文字が表示されます．

d：が省略されると，アセンブラはカレント・ドライブが指定されたものと解釈します．INCLUDE のネストも可能です．

【例】

```
INCLUDE  ABC.ASM
INCLUDE  B：STDIO.H
```

## 5.12／マクロ擬似命令

マクロを定義する **MACRO～ENDM 擬似命令**，マクロ内の名前をローカル指定する **LOCAL 擬似命令**，マクロから抜け出すための **EXITM 擬似命令**，マクロ定義の削除を行う **PURGE 擬似命令**について説明します．

### ● MACRO～ENDM

【書式】

```
名前  MACRO [仮引数，…]
          ⋮
      ENDM
```

【機能】

**MACRO～ENDM 擬似命令**は，マクロ本体を定義し，そのマクロに名前を付けます．マクロの呼び出し（マクロコール）を行う際には，この名前を用いて，

名前 [実引数, …]

とします．実引数，仮引数ともカンマ(,)で区切り，複数の引数の受け渡しが行えます．引数がなければ何も書きません．

アセンブラはマクロコール位置のマクロ名とマクロ本体（ソース・レベル）を置き換えます．これを**マクロ展開**と呼びます．リスティング・ファイルでは，マクロ展開されたステートメントの前に，＋印が付けられます．

マクロ定義部はセグメントの外に置くことができます．

【例】

```
ABC   MACRO   X, Y         ┐
      MOV     AX, X        │
      MOV     BL, Y        ├マクロ定義
        ⋮                  │
      ENDM                 ┘

XYZ   MACRO   MSG          ┐
      DB  ' &MSG'          ├マクロ定義
        ⋮                  │
      ENDM                 ┘

CODE SEGMENT
      ABC     ARRAY,'A'        …マクロコール
        ⋮
DATA SEGMENT
      XYZ     hello            …マクロコール
        ⋮
```

● LOCAL

【書式】

LOCAL 名前 [, …]

【機能】

**LOCAL 擬似命令**は，マクロブロック内で定義されている名前がマクロ展開時に同じ名前で展開されないようにアセンブラに指示をします．これによってアセンブラは，??0000～??FFFF という独自の名前を順次割り当てていきます．

LOCAL 擬似命令はマクロの先頭に置かなければなりません．

【例】

```
ABC   MACRO
      LOCAL   STR,MSG
        ⋮
STR： MOV     AL,BL
        ⋮
MSG   DB   'hello'
      ENDM
```

● EXITM

【書式】

```
EXITM
```

【機能】

**EXITM 擬似命令**は，マクロ内に置かれ，条件擬似命令とともに使用され，マクロ展開を途中で中止することに使われます．

【例】

```
ABC   MACRO   FLAG
        ①
      IFE     FLAG－10
      EXITM  ←――――
      ENDIF
        ②
      ENDM
```

FLAG に 10 が渡されたならば，①の部分はマクロ展開されるが，②の部分は EXITM で抜けるのでマクロ展開されない．

FLAG に 10 以外が渡されたならば，①②の部分は両方ともマクロ展開される

### ● PURGE

【書式】

```
PURGE マクロ名 [, …]
```

【機能】

**PURGE 擬似命令**で指定されたマクロ名のマクロ定義を削除します．

【例】

```
INCLUDE    MACRO.LIB
PURGE      ABC, XYZ  ←── これ以後にABCやXYZというマクロコ
   ⋮                      ールがあっても，マクロ展開されない
```

## 5.13 / 繰り返し擬似命令

マクロアセンブラMASMの繰り返し擬似命令には，**REPT（繰り返し）**，**IRP（不定回数の繰り返し）**，**IRPC（不定回数の文字の繰り返し）**の3種類があります．

繰り返し擬似命令は，擬似命令とENDMで囲まれたブロックを繰り返し展開します．これらの擬似命令REPT，IRP，IRPCは**システムマクロ**と呼ばれ，ユーザがMACRO擬似命令によって定義する**ユーザマクロ**と区別をしています．

システムマクロとユーザマクロは次の点で異なります．

- **MACRO擬似命令で定義されたマクロは，プログラムのどこからでもコールされ，その位置に展開されるのに対して，システムマクロは他からコールされることはなく，定義された位置に展開される**

- **上の理由から，MACRO定義はセグメント外で行われてもよいが，システムマクロはセグメント内で行われなければならない**

● REPT（Repeat）

【書式】

```
REPT 式
     ︙
ENDM
```

【機能】

REPT 擬似命令は，REPT と ENDM で囲まれたブロックを，式で示された回数だけ繰り返しコード（データ）展開します．

● IRP（Indefinite Repeat）

【書式】

```
IRP  仮引数，＜実引数，…＞
     ︙
ENDM
```

【機能】

**IRP 擬似命令**は，＜＞で囲まれた実引数を仮引数に渡すたびに展開し，その結果，実引数の個数だけ IRP と ENDM で囲まれたブロックが繰り返されます．

実引数には，名前（シンボル），文字列，文字定数，数値が指定できます．

● IRPC（Indefinite Repeat Character）

【書式】

```
IRPC  仮引数，文字列
     ︙
ENDM
```

【機能】

文字列の 1 文字を仮引数に渡し，そのたびに展開を繰り返します．したがって，文字列の長さだけ，IRPC と ENDM で囲まれたブロックを繰り返します．

**【例】**

・次の３つの繰り返し擬似命令は，どれも，

```
DB   1
DB   2
DB   3
DB   4
DB   5
```

と展開されます．

```
Y    =    0
     REPT 5
Y    =    Y + 1
     DB   Y
     ENDM
```

---

```
     IRP  Y,<1, 2, 3, 4, 5>
     DB   Y
     ENDM
```

---

```
     IRPC Y, 12345
     DB   Y
     ENDM
```

# 5.14／条件擬似命令

**条件擬似命令**は，一般に次の書式を採ります．

```
IFxxxx ［引数］
     ①
［  ELSE
     ②
          ］
ENDIF
```

引数の評価結果が xxxx で指される条件ステートメントと一致すれば，IF ブロック①がアセンブルされ，一致しなければ ELSE ブロック②がアセンブルされます．また，ELSE 節は省略可能です．

条件は 255 レベルまで入れ子（ネスト）することができますが，１つの IF には１つの ELSE が対応していなければなりません．

xxxx の違いにより，次の 10 種類の IF 擬似命令があります．

【書式】

IF ／ IFE　式

【機能】

式が真（０以外）／偽（０）ならば，IF ブロックがアセンブルされます．

【例】

```
IF    CONST－10 … CONST が 10 以外のとき IF ブロックをアセンブルする
IFE   CONST－10 … CONST が 10 のとき IF ブロックをアセンブルする
```

【書式】

IF1 ／ IF2

【機能】

アセンブラがパス１／パス２実行中ならば，IF ブロックがアセンブルされます．

【書式】

IFDEF ／ IFNDEF　シンボル

【機能】

名前（シンボル）が定義（外部定義も含む）されていれば／いなければ，IF ブロックがアセンブルされます．

【例】

```
ABC  MACRO  DUMY
     IFDEF  DUMY        … XYZ は定義されているので，これが DUMY に渡さ
       ⋮                  れたときは IF ブロックをアセンブルする
     ENDIF                 QQQ は未定義なので，これが DUMY に渡されたと
     ENDM                 きは ELSE ブロックをアセンブルする
       ⋮
XYZ：
     ABC    XYZ
     ABC    QQQ
```

【書式】

IFB ／ IFNB　＜引数＞

【機能】

引数がブランク（ヌル）または空白ならば／でないならば，IF ブロックがアセンブルされます．

引数は＜＞で囲まれなければなりません．

【例】

```
ABC   MACRO   DUMY
      IFB     <DUMY> …  最初のABCのマクロコールではDUMYにヌルが
       ⋮                渡されるのでIFブロックがアセンブルされる
                          次のABCのマクロコールでは，DUMYに'hello'
      ENDIF             が渡されるので，ELSEブロックがアセンブルされる
      ENDM
        ⋮
      ABC
      ABC     hello
```

【書式】

```
IFIDN／IFDIF   <引数1>,<引数2>
```

【機能】

文字列引数1と文字列引数2が，等しい／等しくないならば，IFブロックがアセンブルされます．

引数は＜＞で囲まれなければならなりません．

【例】

```
ABC   MACRO   DREG,SREG
      IFDIF  <DREG>,<SREG> …  引数DREGとSREGが等しければ，
       ⋮                       IFブロックをアセンブルする
      ENDIF
      ENDM
       ⋮
      ABC     AX,AX
      ABC     AX,BX
```

# 5.15 / リスティング擬似命令

アセンブル結果をリスティング・ファイルへ出力する場合，その制御を行うための擬似命令には，次の 14 種類のものがあります．

## ● PAGE

【書式】

```
PAGE  [長さ][, 幅]
PAGE  [+]
```

【機能】

**PAGE 擬似命令**は，リスティング・ファイルの 1 ページのサイズを長さ（行方向），幅（桁：カラム方向）で指定した値に設定します．ディフォルト値は，50 行，80 桁です．

指定できる値は，長さ 10～255 行，幅 60～132 桁までで，この範囲を超えることはできません．

引数なし，または「+」が付けられた PAGE 指定は，新しいページを開始します．つまり，現在のページの終わりにフォームフィード・キャラクタを置きます．ページ番号は＜メジャー＞－＜マイナー＞（例えば 2 － 1 など）の形式で出力されますが，引数なしの PAGE 指定ではマイナー番号を＋1し，「+」指定ではメジャー番号を＋1して，マイナー番号を 1 にリセットします．

【例】

```
PAGE  60,132   … 1ページを 60 行，132 桁に指定
```

## ● TITLE ／ SUBTTL

【書式】

```
TITLE ／ SUBTTL テキスト
```

【機能】

**TITLE 擬似命令**は，各ページの先頭に表示するタイトルを指定します．**SUBTTL 擬似命令**はタイトルの次の行に表示するサブタイトルを指定します．

テキストに使われる文字列は，60文字までの長さでなければなりません．

TITLE擬似命令は，プログラム中に1回指定するだけですが，SUBTTLは何回指定しても，そのたびにサブタイトルの内容が変わっても構いません．

【例】

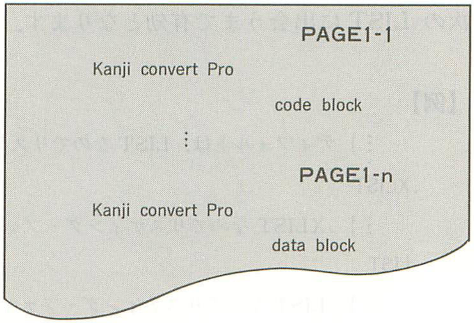
```
TITLE    Kanji convert Pro

   ⋮

SUBTTL   code block

   ⋮

SUBTTL   data block
```




## ●% OUT

【書式】

```
% OUT
```

【機能】

**% OUT 擬似命令**は，指定したテキストをアセンブル中にスクリーンに出力します．これはアセンブルに時間がかかる場合，作業の進行状況をプログラマが知るために使われます．

【例】

```
% OUT   now part 2 assembling
```

## ●.LIST／.XLIST

【書式】

```
.LIST／.XLIST
```

【機能】

**.LIST／.XLIST 擬似命令**は，リスティング・ファイルへリストを出力するかしないかをアセンブラに指示します．

.LISTは，すべての行をそのコードとともにリスティング・ファイルに出力します．.LISTがディフォルトです．

.XLISTは，すべてのリストを出力しません．.XLISTは宣言されてから，次の.LISTに出会うまで有効となります．

【例】

```
        :} ディフォルトは．LISTなのでリスティング・ファイルに出力する
.XLIST
        :} .XLISTなのでリスティング・ファイルに出力しない
.LIST
        :} .LISTなのでリスティング・ファイルに出力する
```

## ●.SFCOND／.LFCOND／.TFCOND

【書式】

.SFCOND／.LFCOND／.TFCOND

【機能】

**.SFCOND／.LFCOND／.TFCOND 擬似命令**は，偽条件ブロックをリスティング出力するかしないかをアセンブラへ指示します．

.SFCONDは偽条件ブロックのリスティングを抑止します．

.LFCONDは偽条件ブロックのリスティングも行います(ディフォルト)．

.TFCONDは，アセンブラの実行時に指定する／Xスイッチ(偽条件ブロックのリスティングを抑止)の設定を反転させます．

【例】

```
0000                     DATA   SEGMENT
                                .LFCOND
                                IFE     0
0000  48 65 6C 6C 6F            DB      'Hello'
                                ELSE
                                DB      'Candy' ┐偽条件ブロック
                                                 も表示される
```

```
                                        ENDIF
                                        .SFCOND
                                        IFE       0
0005   48 65 6C 6C 6F                   DB        'Hello'
                                        ENDIF            ――偽条件ブロック
000A                           DATA     ENDS               は表示しない
                                        END
```

## ●.XALL／.LALL／.SALL

【書式】

.XALL／.LALL／.SALL

【機能】

.XALL／.LALL／.SALL **擬似命令**は，ソース・コード，オブジェクト・コードをリスティング出力するかしないかをアセンブラへ指示します．

.XALL はマクロによって作成されるソース・コードとオブジェクト・コードをリスティングします．.XALL がディフォルトです．

.LALL はコードを生成しないソース行（コメント行）を含むすべてのマクロ展開を展開を行い，リスティングします．ただし，；；で始まるコメントはリスティングしません．

.SALL はマクロによって作成されるすべてのテキスト，およびオブジェクト・コードのリスティングを抑止します．

【例】

```
                               ABC      MACRO
                                                 ;; comment 1
                                                 ;  comment 2
                                        DB       'hello'
                                        ENDM

0000                           DATA     SEGMENT
                                        ABC                 ┐ディフォルトは
0000   68 65 6C 6C 6F      +            DB       'hello'    ┘.XALL
                                        .LALL               ┐
                                        ABC                 │コメントなどを
                           +                                │リスティングするが，
                           +                     ;  comment 2  │;; はリストしない．
                           +                                │
0005   68 65 6C 6C 6F      +            DB       'hello'    ┘
                                        .SALL           ┐リスティングが抑止された
                                        ABC             ┘
000F                           DATA     ENDS
                                        END
```

## ●.CREF／.XCREF

【書式】

```
.CREF
.XCREF [変数, …]
```

【機能】

.XCREF **擬似命令**が宣言されると，それ以後(.CREFに出会うまで)のクロスリファレンスの作成を抑止します．

.XCREFが変数リストを持つ場合，これらの変数はリスティングまたはクロスリファレンス・ファイル内には置くことができません．

# 第6章

## MASM＆ユーティリティ・オペレーション

MASM
MS-LINK
MS-LIB
MS-CREF
DEBUG

## 本章の目的

本章では，MS-DOS 上で機械語プログラムを開発するためのユーティリティを実際の使用例を示して説明します．各ユーティリティについては次のような内容を説明します．

・MASM（マクロアセンブラ）

マクロアセンブラ MASM の起動法，スイッチの使い方，リスティング・ファイルのフォーマットなど，MASM を操作するのに必要な事柄を説明します．

・LINK（リンカ）

LINKはMASMおよび高級言語（MS-Pascal, MS-Fortran, Lattice Cなど）が出力するリロケータブル・オブジェクトをリンクするユーティリティです．LINK の説明では，LINK の起動法，リンカに対して細かい指示を与えるスイッチの使い方について述べます．

・LIB（ライブラリ・マネージャ）

リロケータブル・オブジェクト・レベルでのライブラリ管理をするライブラリ・マネージャ MS-LIB の概要および起動法，さらにライブラリへの操作を示すコマンドキャラクタについて説明します．

・CREF（クロスリファレンサ）

クロスリファレンサ MS-CREF の概要および起動法について説明します．クロスリファレンス・ファイルはデバッグ時に利用されます．

・DEBUG（デバッガ）

プログラムの文法上の誤りは，アセンブラが見つけてくれますが，論理的な誤りは人間が見つけなくてはなりません．これを支援するのが DEBUG です．デバッガは被テスト・プログラムをロード・実行しながらレジスタやメモリの内容を表示したり，修正したりすることができます．

DEBUG の説明では，DEBUG の起動法と基本的な使用例を示し，デバッガの持ついくつかのサブコマンドの使い方を述べます．

# 6.1 / MASM マクロアセンブラ・オペレーション

## 6.1.1 MASMの起動法

MASM の起動法には次の 2 つの方法があります．

### ●方法 1　単独で起動する

A＞MASM ↵

Source filename [.ASM]：ソース・ファイル名 ↵

Object filename [source .OBJ]：オブジェクト・ファイル名 ↵

Source listing　[NUL .LST]：リスティング・ファイル名 ↵

Cross reference [NUL .CRF]：クロスリファレンス・ファイル名 ↵

各プロンプトに対応するファイル名を入力します．↵キーのみ入力すると，[　]で囲まれたディフォルトのファイル名が付けられます．ファイルのタイプ(拡張子)を省略すると，[　]内のタイプが採用されます．各応答の最後に，**6.1.2** で述べる**スイッチ**を設定することができます．

プロンプトに対してセミコロン（；）を入力すると，それ以後のプロンプトは表示されず，ディフォルトのファイル名が採用されます．

NUL とは，ファイルを何も作らないということを意味します．

**【例】**

```
A>MASM
The Microsoft MACRO Assembler
Version 1.20, Copyright (C) Microsoft Inc. 1981,82,83

Source filename [.ASM]: B:TEST
Object filename [TEST.OBJ]: B:TEST
Source listing  [NUL.LST]: B:TEST
Cross reference [NUL.CRF]:

Warning Severe
Errors  Errors
0       0
```

**●方法 2　ファイル名をコマンドラインに与えて起動する**

【例】

A＞MASM B：TEST,，B：TEST；⏎

…B：TEST .ASM をアセンブルして，B：TEST .LST を作成する

## 6.1.2 MASMに入力するファイル名とスイッチ

MASM に入力するファイルは**表 6.1.1** に示す 4 種類があります．

●表6.1.1　MASM に入力するファイル名

| ファイル | 意　　味 |
|---|---|
| ソース・ファイル<br>Source filename [.ASM] | アセンブルするソース・ファイル．タイプを省略すると.ASM となる． |
| オブジェクト・ファイル<br>Object filename [source.OBJ] | アセンブルにより生成されるリロケータブルなオブジェクト．タイプを省略すると.OBJ となる． |
| ソース・リスティング・ファイル<br>Source listing [NUL.LST] | アセンブル結果のリスティング・ファイル．標準タイプは.LST |
| クロスリファレンス・ファイル<br>Cross reference [NUL.CRF] | アセンブル結果のクロスリファレンス・ファイル．標準タイプは.CRF |

MASM には，アセンブラに細かい指示を与えるために，**表6.1.2**に示すような 3 つのスイッチによる指定があります．スイッチは各コマンドラインの終わりに，1 つまたは複数個を並べて書くことができます．

●表6.1.2　MASM のスイッチ

| スイッチ | 機　　能 |
|---|---|
| ／D | パス 1，パス 2 の両方でソース・リスティングを作成する．プログラム中のどこでフェーズ・エラーが起きたかを調べる手掛かりを与える |
| ／O | リスティングを 8 進（Octal）基数で出力する |
| ／X | 偽条件ブロックのリスティング・ファイルへの出力を抑止する |

【例】

A＞MASM　B:TEST,,B:TEST ／ X ; ⏎

…B：TEST．ASMをアセンブルして，B：TEST．LSTを作る．リスティングには偽条件ブロックは出力されない

## 6.1.3 リスティング・ファイルのフォーマット

### ●プログラム・リスティング

リスティングのプログラム部は，アドレス・オフセット，生成されたオブジェクト・コード，ソース・プログラムからなり，それに各種リスティング・シンボルが付加されています．**図6.1.1**にリスティング・ファイルの一部を示します．

図6.1.1　リスティング・ファイル

```
The Microsoft MACRO Assembler                    09-27-85        PAGE      1-1


                                   ABC      MACRO    DUMY          ┐
                                            DB       '&DUMY'       │マクロ定義
                                            ENDM                   ┘

                                   TAIKEI   STRUC                  ┐
0000   41 41 41 41 41              NAME     DB       'AAAAA'       │構造体定義
0005   ????                        BUST     DW       ?             │
0007                               TAIKEI   ENDS                   ┘

= 0200                             CONST    EQU      512

0000                               CODE     SEGMENT  BYTE PUBLIC 'CODE'
                                            ASSUME   CS:CODE,DS:DATA,SS:STACK
                                            EXTRN    MSG:WORD
0000   B8 ---- R                   START:   MOV      AX,DATA
0003   26: 8B 1E 0000 E                     MOV      BX,ES:MSG
0008                               CODE     ENDS

0000                               DATA     SEGMENT  WORD PUBLIC 'DATA'
                                            ABC      hello
0000   68 65 6C 6C 6F            +          DB       'hello'
0005   41 6E 6E 20 20                       TAIKEI   <'Ann',80>
000A   0050

000C                               DATA     ENDS

0000                               STACK    SEGMENT  STACK
0000        32 [                            DW       50 DUP (?)
                  ????
                             ]

0064                               STACK    ENDS

                                            END
```

図中の注記：
- 初期化されない4バイト(2ワード)データ（0005 ???? を指す）
- EQUで定義されている（= 0200 を指す）
- セグメント名（----）
- リンカが値を設定する（R）
- セグメント・オーバライド（26:）
- 外部参照（E）
- マクロ展開された行（+）
- 32Hワード分DUPで確保されている（????）

リスティング・ファイルに示される各リスティング・シンボルの意味を**表6.1.3**にまとめます.

●表6.1.3 リスティング・シンボルの意味

| リスティング・シンボル | 意味 |
|---|---|
| R | Rの左にある項目はリンカが最終的に処理を決定する |
| E | Eの左にある項目は外部参照であることを示す |
| ーーーー | セグメント名/グループ名/セグメント変数であることを示す |
| = | =の左にある値がEQUか=擬似命令で定義されていることを示す |
| nn: | セグメント・オーバーライドであることを示す |
| nn/ | REPまたはLOCKプレフィックスの付いた命令であることを示す |
| [ xx ] | DUP命令で定義されたデータであることを示す |
| + | マクロ展開された行 |
| C | INCLUDEにより取り込まれた行 |

## ●シンボル・テーブル

リスティングのシンボル・テーブルは次の4つの部分に分けて表示されます.**図6.1.2**にシンボル・テーブルの一部を示します.

- ・マクロ
- ・構造体/レコード
- ・セグメント/グループ
- ・シンボル(名前)

図6.1.2　シンボル・テーブル

```
 The Microsoft MACRO Assembler                    09-27-85       PAGE     Symbols
                                                 -1

                                              マクロが占めるブロックサイズ
                                              (1ブロック32バイト)
Macros:
  マクロ名
                  N a m e                       Length

ABC. . . . . . . . . . . . . . .                0001

Structures and records:                          構造体のサイズ（バイト）

 構造体タグ       N a m e                       Width   # fields
                                                Shift   Width    Mask    Initial
          メンバ
TAIKEI . . . . . . . . . . . . . .              0007    0002←メンバ数
  NAME . . . . . . . . . . . . . . .            0000
  BUST . . . . . . . . . . . . . . .            0005
                                                 メンバの構造体内オフセット
Segments and groups:

 セグメント名     N a m e                       Size    align   combine class

CODE . . . . . . . . . . . . . . .              0008    BYTE    PUBLIC  'CODE'
DATA . . . . . . . . . . . . . . .              000C    WORD    PUBLIC  'DATA'
STACK. . . . . . . . . . . . . . .              0064    PARA    STACK

Symbols:

 シンボル(名前)   N a m e                       Type    Value   Attr     外部参照

CONST. . . . . . . . . . . . . . .              Number  0200
MSG. . . . . . . . . . . . . . . .              V WORD  0000    CODE    External
START. . . . . . . . . . . . . . .              L NEAR  0000    CODE

Warning Severe                                          シンボルのあるセグメント
Errors  Errors
0       0
```

# 6.2 / MS-LINK リンカ・オペレーション

## 6.2.1 LINKの起動法

LINK の起動法には次の2つの方法があります.

### ●方法1 単独で起動する

A＞LINK ↵

Object Modules [.OBJ]：オブジェクト・ファイル・リスト名 ↵

Run File [Object .EXE]：ラン・ファイル名 ↵

List File [NUL .MAP]：リスト・ファイル名 ↵

Libraries [.LIB]：ライブラリ・ファイル・リスト名 ↵

各プロンプトに対応するファイル名を入力します. ↵キーのみ入力すると，[ ]で囲まれたディフォルトのファイル名が付けられます. ファイルのタイプ(拡張子)を省略すると，[ ]内のタイプが採用されます. 各応答の最後に, 6.2.2で述べる**スイッチ**を設定することができます.

プロンプトに対してセミコロン(；)を入力すると，それ以後のプロンプトは表示されず，ディフォルトのファイル名が採用されます. **NUL** とは，ファイルを何も作らないということを意味します.

【例】

```
A>LINK

   Microsoft Object Linker V2.2(830822)
(C) Copyright 1982, 1983 by Microsoft Inc.

Object Modules [.OBJ]: B:MMODUL+B:SMODUL1+B:SMODUL2
Run File [A:MMODUL.EXE]: B:MMODUL
List File [NUL.MAP]:
Libraries [.LIB]: MYLIB
```

ドライブBの MMODUL .OBJ, SMODUL 1 .OBJ, SMODUL 2 .OBJ をこの順序でリンクし，さらにドライブAの MYLIB .LIB をリンクしてドライブBに MMODUL .EXE を作成する

**●方法 2　ファイル名をコマンドラインに与えて起動する**

A＞LINK　オブジェクト・リスト［, ラン・ファイル］［, リスト・ファイル］

［, ライブラリ・リスト］［／スイッチ］↵

ファイルは方法1の場合と同様のものです．ファイル名を省略してディフォルト値（方法1の［　］の中のファイル名）を用いるときは，単にカンマ（，）のみを置きます．後にファイル名を置く必要がない場合は，最後にセミコロン（；）を置きます．

【例】

A＞LINK B：TEST，B：TEST,，MYLIB↵

…B：TEST .OBJをロードし，MYLIB .LIBとリンクして
TEST .EXEという実行可能ファイルを作成する

**●方法 3　応答ファイル名をコマンドラインに与えて起動する**

A＞LINK　@filename↵

filenameで示される応答ファイルには，LINKに応答する4種類のファイル名をあらかじめエディタで作成しておきます．filenameのファイルのタイプは何であっても構いません．

たとえば，次のような応答ファイル LINKF をエディタで作成したとします．

```
B：TEST↵
B：TEST↵
↵
MYLIB↵
```

←応答ファイル
LINKF

このとき，

A＞LINK　@LINKF↵

とすると，B：TEST .OBJ が MYLIB .LIBとリンクされ B：TEST .EXE ファイルが作成されます．

### 6.2.2 LINKに入力するファイル名とスイッチ

LINKに応答するファイル名は**表6.2.1**に示す4種類があります．

●表6.2.1　LINKに入力するファイル

| ファイル | 意　味 |
|---|---|
| オブジェクト・ファイル<br>Object Modules [.OBJ] | リンクするオブジェクト・モジュール．<br>ディフォルト・タイプは .OBJ．オブジェクトが複数ある場合は，プラス(+)，空白(␣)で区切る．リンカは並べられた順に同じクラス名のセグメントをまとめてロードする |
| ラン・ファイル<br>Run File [Object.EXE] | 実行可能ファイル．<br>タイプ指定は無視され，必ず.EXEが付けられる |
| リスト・ファイル<br>List File [NUL.MAP] | ロードされたモジュールの各セグメントに対するエントリをマップしたリスト・ファイル |
| ライブラリ・ファイル<br>Libraries [.LIB] | ライブラリ・ファイルはライブラリ・マネージャLIBにより作成されたものでなければならない．ライブラリが複数ある場合は，プラス（+）または空白(␣)で区切る |

LINKには，リンカに細かい指示を与えるために，**表6.2.2**に示すような7種類のスイッチによる指定があります．スイッチは各コマンドラインの終わりに，1または複数を並べて書くことができます．

スイッチ名として，頭文字1文字から完全なスイッチ名の長さまでの任意の文字列で指定できます．たとえば，／HIGHというスイッチは，／Hと書いても，／HI，／HIGHと書いても構いません．

●表6.2.2　LINKのスイッチ

| スイッチ | 機　能 |
|---|---|
| /DSALLOCATE | すべてのデータ(DGROUP)をデータ・セグメントのハイエンドにロードする |
| /HIGH | ラン・ファイルはメモリ内のできるだけ高い位置に配置される |
| /LINENUMBERS | リスト・ファイルに出力されるリストの各行に行番号とアドレスを付ける |
| /MAP | 入力モジュール内で定義されたすべてのPUBLIC（グローバル）シンボルをリストする |
| /PAUSE | LINKの処理を一時中止し，ディスク交換の指示を出す．各種ファイルが別のディスク上にあるときに便利である |
| /STACK：値 | 値で示される大きさのスタックを取る．値は65536バイトまでの正の10進数．値が512バイト未満ならば512となる |
| /NO | ディフォルトのライブラリ検索を行わないように指示する．ある種のコンパイラはライブラリをリンク時に指定しなくても自動的にリンクするので，それを行わないようにする |

# 6.3／MS-LIB ライブラリ・マネージャ・オペレーション

## 6.3.1 ライブラリ・マネージャとは

各プログラムから共通に利用できるルーチンを再利用できるようにすることはプログラムの生産性を向上させます．

すでに説明した**マクロ**やファイルの取り込みを行う**INCLUDE擬似命令**は，ソース・レベルでのルーチンの再利用を支援するものです．

さらに効率のよい方法として，アセンブル（または，コンパイル）により生成されたリロケータブル・オブジェクト（.OBJ）を再利用することも考えられます．リロケータブル・オブジェクトのレベルでライブラリとして構築していくことができればプログラミングをする上で好都合といえます．特に異なるソース（MASMとC, MASMとPascalなど）のプログラムをリンクする場合には重要なことです．

このリロケータブル・オブジェクトのレベルでライブラリを管理するのが**ライブラリ・マネージャ（LIB）**です．

さて，ライブラリ・マネージャについて説明する前に，オブジェクト・ファイルとオブジェクト・モジュールという用語についてここで定義しておくことにします．

**オブジェクト・ファイル**とはアセンブル（またはコンパイル）により生成されたリロケータブル・オブジェクトです．たとえば，

```
B：SMODUL1.OBJ
```

のようにドライブ名とファイルタイプ（.OBJ）を持つものです．

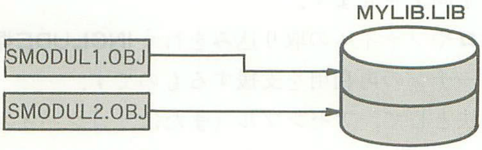
一方，**オブジェクト・モジュール**とはオブジェクト・ファイルをLIB（ライブラリ・マネージャ）によりライブラリとしたときのモジュールを指します．ドライブ名とファイルタイプを取ったSMODUL 1がライブラリ内での名前となります．

用語の定義が終わったところで，ライブラリ・マネージャの機能について説

明をすることにします．

**●ライブラリの新規作成**

一般に，ユーザ・ライブラリは何もない状態から新規に作成されていきます．そこで，SMODUL 1.OBJ と SMODUL 2.OBJ というリロケータブル・オブジェクト（4章4.12の図4.19，図4.20のようなもの）を MYLIB.LIBとして登録します．



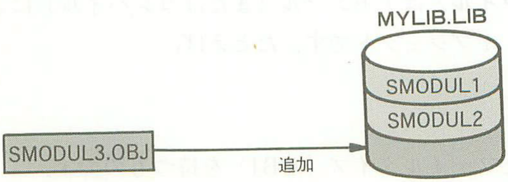

```
A>LIB

   Microsoft Library Manager V1.02
(C) Copyright 1981 by Microsoft Inc.

Library File: MYLIB
Library does not exist. Create?Y←MYLIB.LIB は新規作成なので問合せがある．'Y' or 'y' と答える
Operations: +B:SMODUL1+B:SMODUL2←空のライブラリへ２つのモジュールを登録する
List file:
```

**●ライブラリへの追加**

新規作成されたライブラリ（MYLIB .LIB）へ新しいモジュールSMODUL3 .OBJ を追加します．基本的にはライブラリへの追加も，ライブラリの新規作成も同じ操作です．唯一違うのは，新規作成においてはライブラリ・ファイルが存在しないので，LIB からの問い合わせに対して，“Y”と答えることだけです．




```
A>LIB

   Microsoft Library Manager V1.02
(C) Copyright 1981 by Microsoft Inc.

Library File: MYLIB
Operations: +B:SMODUL3←────────追加
List file:
```

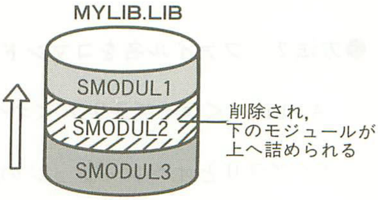
**●ライブラリからの削除**

ライブラリ MYLIB .LIB から不要になったモジュール SMODUL2 を削除します．SMODUL 2 のあった位置には，下位にあったSMODUL 3 が詰められるので，削除を繰り返してもライブラリ内には空きが生じないようになっています．

```
A>LIB

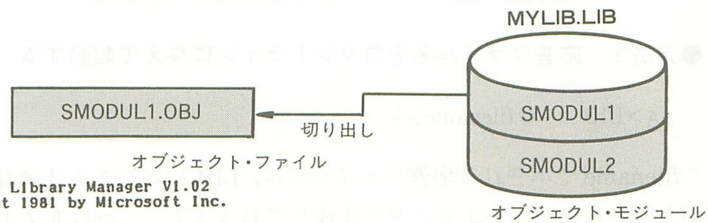
   Microsoft Library Manager V1.02
(C) Copyright 1981 by Microsoft Inc.

Library File: MYLIB
Operations: -SMODUL2 ←―――――― 削除
List file:
```




**●ライブラリからモジュールをオブジェクト・ファイルとして切り出す**

ライブラリ MYLIB .LIB 内のモジュール SMODUL 1 をオブジェクト・ファイル（SMODUL 1 .OBJ）として切り出します．このときライブラリ内の SMODUL 1 はそのままです．




```
A>LIB

   Microsoft Library Manager V1.02
(C) Copyright 1981 by Microsoft Inc.

Library File: MYLIB
Operations: *SMODUL1 ←―――――― 切り出す
List file:
```

## 6.3.2　LIBの起動

LIB の起動法には次の 3 つの方法があります．

**●方法 1　単独で起動する**

A＞LIB↵

Library File：ライブラリ・ファイル名↵

Operations：オペレーション↵

List File［NUL．MAP］：リスト・ファイル↵

各プロンプトに対応するファイル名またはオペレーションを入力します．↵キーのみ入力すると，［ ］で囲まれたディフォルトのファイル名が付けられます（**6.3.3**参照）．プロンプトに対してセミコロン（；）を入力すると，それ以後のプロンプトは表示されず，ディフォルトのファイル名が採用されます．

**●方法2　ファイル名をコマンドラインに与えて起動する**

A>LIB ＜ライブラリ＞ ＜オペレーション＞，リスト・ファイル ↵

ライブラリとオペレーションの区切りは，コマンドキャラクタのプラス（＋），マイナス（－），アスタリスク（＊）のいずれかで分離しなければなりません．

【例】

A>LIB　MYLIB＋B：SMODUL 3；↵

…MYLIB .LIBにドライブBのSMODUL 3 .OBJを追加する

**●方法3　応答ファイル名をコマンドラインに与えて起動する**

A>LIB　@filename ↵

filenameで示される応答ファイルには，LIBに応答する3種類のファイル名のものをあらかじめエディタで作成しておきます．これは**6.2.1**のLINKのところで説明したのと同じ種類のものです．

### 6.3.3　LIBに入力するファイル名とコマンドキャラクタ

LIBに応答するファイル名（オペレーションも含む）には**表6.3.1**に示す3種類があります．

●表6.3.1　LIBに入力するファイル名

| ファイル | 意　味 |
|---|---|
| ライブラリ・ファイル<br>Library File： | 新規または修正するライブラリ・ファイル．<br>ディフォルト・タイプは .LIB．ライブラリが新規の場合（ディスク上にその名前のライブラリ・ファイルがないとき）だけ，LIBは問い合わせてくるので，“Y”または“y”と答える．それ以外の文字を入力すると処理は中断 |

| | |
|---|---|
| | される |
| オペレーション<br>Operations： | コマンドキャラクタ（＋，－，＊）とそれに続くモジュール名を並べたもので，各モジュールやオブジェクト・ファイルに対し，どのような処理をするのかをLIBに指示するためのもの．ディフォルト・タイプは .OBJ.<br>↵キーのみ入力するとライブラリの整合性のチェックのみを行う |
| リスト・ファイル<br>List File： | ライブラリの中にあるモジュールのPUBLICシンボルのクロスリファレンス・リスティングを収めるファイル |

オペレーションに書くコマンドキャラクタは表6.3.2に示す５種類があります．

●表6.3.2　コマンドキャラクタ

| キャラクタ | 機　能 |
|---|---|
| ＋ | ライブラリの最後部にオブジェクト・ファイルを追加する |
| － | ライブラリから１つのモジュールを削除する |
| ＊ | ライブラリからモジュールを切出しオブジェクト・ファイルを作る |
| ； | 残りのプロンプトに対し，ディフォルトの応答をする |
| ＆ | コマンド行に書ききれなくなったときに次の行へ継続する |

次にオペレーションの記述例をいくつか示します．

【例】

A＞LIB　MYLIB－DISP＋B：DISP；↵

…MYLIBからDISPを削除し，新しいDISP .OBJを追加する．
DISPのバージョンを更新したことになる

A＞LIB　MYLIB；↵

…MYLIBの整合性のチェックのみを行う

A＞LIB　MYLIB＋ B：DISP－SMODUL1；↵

…MYLIBにDISP .OBJを追加し，SMODUL１を削除する

### 6.3.4 ライブラリのリスティング・ファイル

ライブラリ MYLIB .LIB のリスティング・ファイルを図6.3.1に示します．モジュール名（SMODUL1, SMODUL 2）とPUBLICシンボル（DISP, KEYIN）がリストされていることがわかると思います．リンクされるメイン・モジュールからPUBLICシンボルのDISPやKEYINでエントリされます．

▼図6.3.1　ライブラリのリスティング・ファイル

```
A>TYPE B:MYLIB.LST

DISP..SMODUL2  KEYIN.SMODUL1

SMODUL1 (Length 0005H bytes)  ←モジュール名とサイズ

   KEYIN  ←モジュール内で定義されているPUBLICシンボル

SMODUL2 (Length 0022H bytes)

   DISP
```

ライブラリMYLIB.LIBをメイン・モジュールMMODUL.OBJとリンクするには次のようにします．これを第4章4.12の図4.21と比較してみてください．

```
A>LINK

   Microsoft Object Linker V2.2(830822)
(C) Copyright 1982, 1983 by Microsoft Inc.

Object Modules [.OBJ]: B:MMODUL
Run File [A:MMODUL.EXE]: B:MMODUL
List File [NUL.MAP]:
Libraries [.LIB]: MYLIB
```

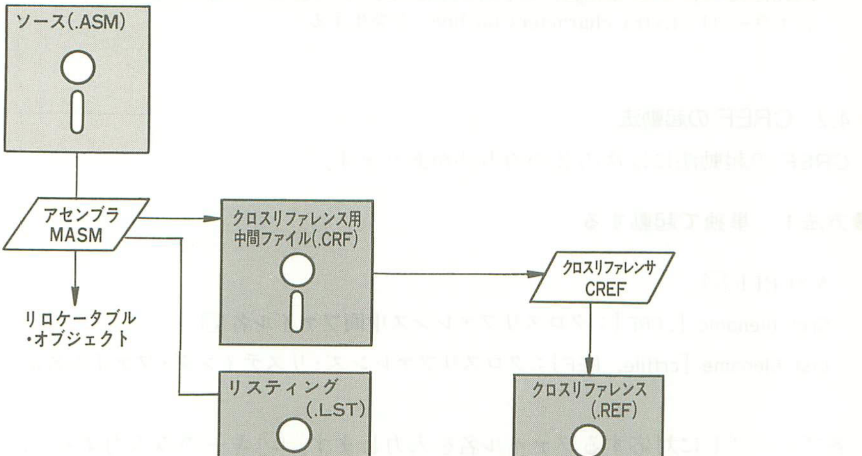
# 6.4／MS-CREF クロスリファレンサ・オペレーション

## 6.4.1 CREF（クロスリファレンサ）とは

クロスリファレンス・ファイルはアセンブリ言語ソース・プログラム中のすべてのシンボルをアルファベット順に並べ，そのシンボルが定義されている行と参照している行をすべてピックアップして表にしたものです．行番号はリスティング・ファイル（.LST）中に表記されているものです．

**図6.4.1**にMS-CREF(**クロスリファレンサ**)を用いてクロスリファレンス・ファイル作成の概要を示します．

図6.4.1　CREFの概要



クロスリファレンス・ファイルを作る場合は，まずMASMによるアセンブル時に，**リスティング・ファイル（.LST）**と**クロスリファレンス用中間ファイル（.CRF）**を作成するかというアセンブラの問いに作成すると答えます．

.CRFファイルを作成すると答えたときだけ，.LSTファイルに行番号が付加されます．.CREFファイルをクロスリファレンサCREFにかけ，**クロスリファレンス・ファイル（.REF）**を作ります．クロスリファレンス・ファイル中のシャープ（#）記号の付いた行は，シンボルがその行で定義されていることを示します．

注）MASM ver 1.20ではクロスリファレンス（.CRF）を出力する際に，次のようなバグがあります．
・パラメータを省略したSEGMENT擬似命令に対して，エラー（25：Not proper align／combine type）が発生する
・Define文のオペランドに長い文字列と複数の式（，で区切ったもの）が重なったときにエラー（1：Extra characters on line）が発生する

### 6.4.2　CREFの起動法

CREFの起動法には次の2つの方法があります．

**●方法1　単独で起動する**

A>CREF ⏎
Cref filename [.CRF]：クロスリファレンス中間ファイル名 ⏎
List filename [crffile. REF]：クロスリファレンス・リスティング・ファイル名 ⏎

各プロンプトに対応するファイル名を入力します．⏎キーのみ入力すると，［　］で囲まれたディフォルトのファイル名が付けられます．

プロンプトに対してセミコロン（；）を入力すると，それ以後のプロンプトは表示されず，ディフォルトのファイル名が採用されます．

**【例】**

```
A>CREF
Cref filename [.CRF]: B:CFILE
List filename [CFILE.REF]: B:CFILE
```

この例では，アセンブルにより作成されているリスティング中間ファイルCFILE .CRFからリスティング・ファイルCFILE .REFを作ります.

**●方法 2　ファイル名をコマンドラインに与えて起動する**

A>CREF クロスリファレンス中間ファイル名，クロスリファレンスリスティング・ファイル名↵

ファイルは方法1の場合と同様なもので，ディフォルト解釈も同じです.

【例】

A>CREF　B：ABC；↵

…ドライブBのABC. CRFをリスティング・ファイルABC. REFに変換する

### 6.4.3　クロスリファレンス・ファイル

リスティング・ファイル（.LST）とクロスリファレンス・ファイル（.REF）を図6.4.2，図6.4.3に示します.

▼図6.4.2　リスティング・ファイル（.LST）

```
┌行番号
↓
 1                                              ; --------------------------
 2                                              ; *       main module       *'
 3                                              ; --------------------------
 4                                              ;
 5                                              PGROUP  GROUP    CODE
 6                                              DGROUP  GROUP    DATA
 7
 8       0000                                   DATA    SEGMENT  WORD PUBLIC 'DATA'
 9       0000   48 69 74 20 6B 65               MSG     DB       'Hit key(1-9)!$'
10              79 28 31 2D 39 29
11              21 24
12       000E                                   DATA    ENDS
13
14       0000                                   CODE    SEGMENT  BYTE PUBLIC 'CODE'
15                                                      ASSUME   CS:PGROUP,DS:DGROUP,SS:
                                         STACK
16                                                      EXTRN    KEYIN:NEAR,DISP:NEAR
17
18       0000   B8 ---- R                       START:  MOV      AX,DATA      ; DS set
19       0003   8E D8                                   MOV      DS,AX
20
21       0005   B4 09                                   MOV      AH,9
22       0007   BA 0000 R                               MOV      DX,OFFSET MSG
23       000A   CD 21                                   INT      21H
24       000C   E8 0000 E                               CALL     KEYIN
25       000F   E8 0000 E                               CALL     DISP
26       0012   B4 4C                                   MOV      AH,4CH       ; jmp MS-DO
                                         S
```

```
27        0014  CD 21                             INT     21H
28        0016                          CODE      ENDS
29
30        0000                          STACK     SEGMENT STACK
31        0000       20 [               DW        32 DUP(?)
32                          ????
33                                 ]
34
35        0040                          STACK     ENDS
36
37                                                END     START
```

▼図6.4.3 クロスリファレンス・ファイル（.REF）

```
 Symbol Cross Reference                    (# is definition)      Cref-1

CODE . . . . . . . . . . . . . . .      5     14#     14      28

DATA . . . . . . . . . . . . . . .      6      8#      8      12      18
DGROUP . . . . . . . . . . . . . .      6     15
DISP . . . . . . . . . . . . . . .     16#    25

KEYIN. . . . . . . . . . . . . . .     16#    24

MSG. . . . . . . . . . . . . . . .      9#    22

PGROUP . . . . . . . . . . . . . .      5     15

STACK. . . . . . . . . . . . . . .     15     30#     35
START. . . . . . . . . . . . . . .     18#    37
```

（DISP の 16# を指して）定義された行であることを示す

# 6.5／DEBUG デバッガ・オペレーション

## 6.5.1 DEBUGの起動とサブコマンドの種類

DEBUGの起動法には次の2つの方法があります．

### ●方法1　DEBUGを起動させると同時に，被テスト・ファイルをロードする

A>DEBUG　ファイル名　[引数]↵

DEBUGを起動させ，ファイル名で表示されるファイルをメモリにロードします．デバッグの対象となるファイルは .EXE および .COM のタイプのものです．引数はプログラム・セグメント・プリフィックス（PSP）のFCBにセットされます．これはコマンドラインで

A>ファイル名　引数↵

として実行した場合と同じ環境を作ります．

### ●方法2　DEBUGのみを起動する

A>DEBUG↵

この方法で起動させた場合，後述するNコマンド，Lコマンドによって被テスト・ファイルをロードすることができます．この方法はすでにメモリ上にあるデータを修正したり，移動したり，内容のダンプをしたりするのによく使われます．

以上の方法によりDEBUGを起動すると，プロンプト"−"が表示され，サブコマンドの入力待ちとなります．

サブコマンドの一覧を次のページの**表6.5.1**に示します．

●表6.5.1　DEBUGのサブコマンド一覧

| サブコマンド | 機　　　能 |
|---|---|
| G (Go) | 被テスト・プログラムの実行 |
| T (Trace) | トレース |
| A (Assemble) | ライン・アセンブル |
| U (Unassemble) | 逆アセンブラ |
| R (Register) | レジスタ内容の表示／変更 |
| D (Dump) | 16進ダンプ |
| E (Enter) | データ・セット |
| F (Fill) | データ・フィル |
| M (Move) | ブロック転送 |
| C (Compare) | 指定領域の比較 |
| S (Search) | 指定領域の検索 |
| N (Name) | ファイル名の設定（セット） |
| L (Load) | ファイルの呼び出し（ロード） |
| W (Write) | ファイルへの書き込み（ライト） |
| H (Hex) | 16進加減算の計算 |
| I (Input) | ポート入力 |
| O (Output) | ポート出力 |
| Q (Quit) | デバッガを抜けDOSに戻る |

## 6.5.2　DEBUGの使用例

DEBUGを用いて被テスト・ファイルをテストする例を示します．テスト・ファイルは第3章 **3.2** で示したメモリを0クリアするプログラムとして，作業例を図 **6.5.2** に示します．

▼図6.5.2　DEBUGの作業例

```
①  A>DEBUG B:DRILL2.COM
    -R
②  AX=0000  BX=0000  CX=0010  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
    DS=1180  ES=1180  SS=1180  CS=1180  IP=0100   NV UP DI PL NZ AC PE NC
    1180:0100 BB8000        MOV     BX,0080
    -UCS:100,10F
    1180:0100 BB8000        MOV     BX,0080
    1180:0103 B98000        MOV     CX,0080
③  1180:0106 C60700        MOV     BYTE PTR [BX],00
    1180:0109 43            INC     BX
    1180:010A E2FA          LOOP    0106
    1180:010C B44C          MOV     AH,4C
    1180:010E CD21          INT     21
    -ECS:10E
④  1180:010E  CD.CC
⑤  -FCS:80,FF,1A
    -DCS:80,FF
    1180:0080  1A 1A 1A 1A 1A 1A 1A 1A-1A 1A 1A 1A 1A 1A 1A 1A   ................
    1180:0090  1A 1A 1A 1A 1A 1A 1A 1A-1A 1A 1A 1A 1A 1A 1A 1A   ................
⑥  1180:00A0  1A 1A 1A 1A 1A 1A 1A 1A-1A 1A 1A 1A 1A 1A 1A 1A   ................
```

```
     1180:00B0  1A 1A 1A 1A 1A 1A 1A 1A-1A 1A 1A 1A 1A 1A 1A 1A   ................
     1180:00C0  1A 1A 1A 1A 1A 1A 1A 1A-1A 1A 1A 1A 1A 1A 1A 1A   ................
     1180:00D0  1A 1A 1A 1A 1A 1A 1A 1A-1A 1A 1A 1A 1A 1A 1A 1A   ................
     1180:00E0  1A 1A 1A 1A 1A 1A 1A 1A-1A 1A 1A 1A 1A 1A 1A 1A   ................
     1180:00F0  1A 1A 1A 1A 1A 1A 1A 1A-1A 1A 1A 1A 1A 1A 1A 1A   ................
⑦   -G=CS:100

     AX=4C00  BX=0100  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
     DS=1180  ES=1180  SS=1180  CS=1180  IP=010E   NV UP DI PL NZ AC PE NC
     1180:010E CC            INT     3
     -DCS:80,FF
     1180:0080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
     1180:0090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
     1180:00A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
⑧   1180:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
     1180:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
     1180:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
     1180:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
     1180:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
⑨   -Q
```

①DEBUGの起動と被テスト・ファイルのロード
②レジスタの内容表示
③逆アセンブル
④プログラム最後のMS-DOSへ戻る部分に"INT 3"を入れる
⑤0クリアする領域を1Aで満たす
⑥満たした内容を確認
⑦プログラムの実行
⑧実行結果の確認
⑨DEBUGから抜ける

DEBUGの使用に際しては次の点に注意してください

## ●アドレスの指定とディフォルト値

DEBUGのコマンドでアドレスの指定をする場合は，セグメント値とオフセット値を用いて，

1180：0100

または，

CS：0100

などと表します．セグメント・レジスタ名としてCS，DS，SS，ESが使用できます．しかし，多くの場合，セグメント値またはセグメント・レジスタ名を省略することができます．省略時のディフォルト値は，コマンドによって次のようにCSとDSに分けられます．

CSの値をディフォルト値とするコマンド　G，L，T，U，W

DSの値をディフォルト値とするコマンド　Qを除いた上記以外のコマンド

## ●ブレーク・ポイントの設定

プログラムの最後は多くの場合，ファンクションコールによってMS-DOSに戻るように作られています．そのため，そのままDEBUGを使ってGコマンドを行うと“Program terminated normally”でプログラムの実行を終了しますが，再度Gコマンドにより実行するとMS-DOSに戻ってしまったり，正常に動作しないことがあります．

実行後，DEBUGから抜けないようにするには，“INT 3”という割り込みをプログラムの終わりに入れておきます．INT 3割り込みは，ブレーク・ポイント用のもので，コードはCCです．

## ●レンジ

多くのコマンドにおいて，アドレスの範囲を指定するのに，2つの方法が用いられます．1つは開始アドレス，終了アドレスという形式，もう1つは開始アドレス，L値という形式です．L値は開始アドレスから相対的な量を示します．したがって次の2つの指定は同じ内容を表しています．

CS：100, 11F…コード・セグメントのオフセット100H～11FH

CS：100, L20…コード・セグメントのオフセット100H～20Hバイト分

この2種類のアドレス範囲の指定方法を以後**レンジ**と呼ぶことにします．レンジの開始アドレスが省略されると，前にコマンドを実行したときの最終アドレスが採用され，レンジの終了アドレスまたはL値が省略されると80Hとして処理されます．ただし，Uコマンドのときは20Hとして処理されます．

数値の先頭の0は省略することができます．たとえば000FはただのFと書いても構いません．

## ●その他

パラメータの区切りはスペースまたはカンマ（,）により行います．

サブコマンドの中止はCTRL＋C，フリーズはCTRL＋Sにより行います．

### 6.5.3 DEBUG のサブコマンド

DEBUG の 18 種類のサブコマンドの使い方を説明します．

#### ● A（Assemble／アセンブル）

【書式】 A［アドレス］

キーボードから入力される 8086 ニーモニックをライン・アセンブルし，アドレス番地以後に機械語コードを格納していきます．アセンブル作業を中止するには ↵ キーのみ入力します．数値はすべて 16 進とみなされるので，H を付けてはいけません．

プリフィックス・ニーモニックはオペコードの前または前の行に置きます．セグメント・オーバーライド・ニーモニックは CS：，DS：，SS：，ES：です．FAR リターンのニーモニックは RETF です．

DB，DW 擬似命令によりデータを定義することもできます．

【例】

```
-A100  ←───────────────────────── CS：100よりアセンブル結果を格納する
1180:0100 MOV   AX,[BP+SI]
1180:0102 MOV   BYTE PTR [BX], 0
1180:0105 ES:
1180:0106 MOV   AX,[BX]
1180:0108 JMP   102
1180:010A JMP   NEAR 500
1180:010D RETF
1180:010E DB    'Hello',0D,0A
1180:0115 ↵  ←────────────────────── アセンブルの終わり
```

#### ● C（Compare／比較）

【書式】 Cレンジ，アドレス

レンジで指定されたメモリ領域とアドレスから始まる同じサイズの領域を比較し，違いを表示する

【例】

－C 100，1 FF，300…DS：100～DS：1FF のメモリと DS：300～DS：3FF
　　のメモリブロックの比較を行う

## ● D（Dump／ダンプ）

【書式】　D［レンジ］

レンジによって指定された範囲のメモリの内容を16進ダンプします．

【例】

－D0, 1F…DS：0～1FHをダンプする

－D　　…直前のコマンドの最終アドレスから80H分ダンプする

－DCS：0, 1F…コード・セグメントのCS：0～1FHをダンプする

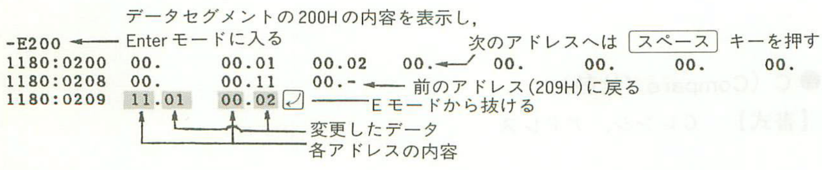
－D10, L50…データ・セグメントのDS：10H～50H分をダンプする

## ● E（Enter／データ・セット）

【書式】　Eアドレス［, リスト］

指定されたアドレスの内容を表示し，入力要求を促します．データを変更しない場合，または次のアドレスに進む場合はスペースキーを押すと次のアドレスの内容が表示され，マイナス（－）を押すと前のアドレスに戻ります．Eコマンドのモードから抜け出すには⏎キーを押します．コマンドラインにリストを付けて入力すると，アドレスの内容はリストで置き換えられますが，内容表示は行われません．

【例】




## ● F（Fill／データ・フィル）

【書式】　Fレンジ，リスト

レンジで指定された範囲のメモリ内容をリストで満たします．

【例】

－F100, 1FF, 1A…データ・セグメントの100H～1FFHを1AHで満たす

−F100, 1FF, 'ABC'…データ・セグメントの100H～1FFHを'ABCで満たす

### ● G（Go／実行）

【書式】　G ［=開始アドレス］［, ブレーク・ポイント, …］

開始アドレスよりプログラムの実行を開始します．開始アドレスを明示するために，必ず"="を付けなければなりません．開始アドレスが省略されると現在のインストラクション・ポインタ（IP）から開始されます．ブレーク・ポイントは最大10個まで設定できます．いずれかのブレーク・ポイントに到達すると，すべてのブレーク・ポイントは解除されます．ブレーク・ポイントには"INT 3"がセットされています．

なお，ブレーク・ポイントは各オペコードの先頭バイトを示していなければなりません．

【例】

−G…現在のインストラクション・ポインタから実行する

−G=0, 108…CS：0から実行し，ブレークポイントを108Hに設定する

−G, 110…現在のインストラクション・ポインタから実行し，ブレークポイントを110Hに設定する

### ●H（Hex／16進計算）

【書式】　H値1，値2

値1＋値2と値1－値2を計算し，表示します．

【例】

```
-H1FF 1
0200   01FE
 |      |________________________ 減算
 |_______________________________ 加算
```

### ● I（Input／ポート入力）

【書式】　Iポート番号

ポート番号で示されたポートから1バイト入力し，表示します．

### ●L（Load／プログラムのロード）

【書式】　L［アドレス［，ドライブ，開始相対セクタ番号，セクタ数]]

Nコマンドでセットされたファイルをメモリにロードします．.COMや．EXEのような実行可能ファイルはリンカにより決められたアドレスにロードされます．それ以外のファイルは指定されたアドレスにロードされ，アドレスが省略されているとCS：100にロードされます．

"ドライブ，開始相対セクタ番号，セクタ数"が指定されると，FBCにセットされているファイルではなく，ドライブで示されるディスクのセクタを直接，指定された範囲で読み込みます．アドレスが省略されるとCS：100とみなされます．ドライブ番号はA＝0，B＝1…です．

読み込まれたバイト数はBX：CXに与えられます．

【例】

```
－NB：MAIN .EXE

－L                    …ドライブBのMAIN.EXEファイルをロードする

-LDS:200,1,5,1 ←――――ドライブBの第5セクタから1セクタ分をDS：200に読み込む
-DDS:200
116C:0200  53 45 47 31 20 20 20 20-41 53 4D 20 00 00 00 00   SEG1    ASM ....
116C:0210  00 00 00 00 00 00 0C 42-41 0B 02 00 D5 00 00 00   .......BA...U...
116C:0220  49 4E 4B 45 59 20 20 20-41 53 4D 20 00 00 00 00   INKEY   ASM ....
116C:0230  00 00 00 00 00 00 83 9E-3B 0B 03 00 3C 01 00 00   ........;...<...
116C:0240  45 58 31 20 20 20 20 20-45 58 45 20 00 00 00 00   EX1     EXE ....
116C:0250  00 00 00 00 00 00 62 2D-41 0B 04 00 70 02 00 00   ......b-A...p...
116C:0260  44 52 49 4C 4C 31 20 20-41 53 4D 20 00 00 00 00   DRILL1  ASM ....
116C:0270  00 00 00 00 00 00 95 4A-47 0B 05 00 94 01 00 00   .......JG.......
```

### ●M（Move／ブロック転送）

【書式】　Mレンジ，アドレス

レンジで指定されるメモリ・ブロックを指定されたアドレス以後にブロック転送します．2つの領域が重なっていても正常な転送が行えます．

### ●N（Name／ファイル名の設定）

【書式】　Nファイル名1［，ファイル名2…］

FCB（ファイル・コントロール・ブロック：PSPの5CH以後がディフォルト）にファイル名を格納する．これには次のような2つの目的があります．1

つはＬコマンド，Ｗコマンドにより被テストプログラムをロード及びセーブするため．もう１つは被テストプログラムがコマンドラインで実行されたときにパラメータを受け取るような場合に，DEBUG でこれを行えるようにするためです．

【例】

```
-NB:DRILL2.COM ABC.TXT
-D0050,009F
1180:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 02 44 52 49   M!K.........DRI
1180:0060  4C 4C 32 20 20 43 4F 4D-00 00 00 00 00 41 42 43   LL2  COM.....ABC
1180:0070  20 20 20 20 20 54 58 54-00 00 00 00 00 F9 C3 F9        TXT.....yCy
1180:0080  14 42 3A 44 52 49 4C 4C-32 2E 43 4F 4D 20 41 42   .B:DRILL2.COM AB
1180:0090  43 2E 54 58 54 0D 00 00-00 00 00 00 00 00 00 00   C.TXT...........
```

FCB

第１パラメータは5CHから

第２パラメータは6CHから

コマンドラインに入力されたパラメータの全文字数

入力されたパラメータのそのままのイメージ

● O（Output／ポート出力）

【書式】　Oポート番号，データ

ポート番号のポートにデータを出力します．

● Q（Quit／DEBUG の終了）

【書式】　Q

DEBUG を終了し，MS-DOS へ戻ります．

●R（Register／レジスタの内容表示と変更）

【書式】　R［レジスタ名］

レジスタ名がなければ，すべてのレジスタ，フラグの内容を表示します．レジスタ名があれば，そのレジスタの内容を表示し，値の変更を求めてきます．何も変更しない場合は，↵キーのみを入力します．

使用できるレジスタ名は次の通りです．

AX，BX，CX，DX

SP，BP，SI，DI，

DS，ES，SS，CS

IP，PC，F

フラグの内容を変更したい場合は，次の名前を用いてセット/リセットします．

●表6.5.2 フラグ名とセット／リセット

| フラグ名 | セット | リセット |
|---|---|---|
| Overflow | OV | NV |
| Direction | DN（減少） | UP（増加） |
| Interrupt | EI （許可） | DI （不許可） |
| Sign | NG（負） | PL （正） |
| Zero | ZR | NZ |
| Auxiliary Cary | AC | NA |
| Parity | PE （偶数） | PO （奇数） |
| Carry | CY | NC |

【例】

```
-R ←──現在のレジスタの内容を表示する
AX=0000  BX=0000  CX=0010  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000 ┐レジスタの
DS=1180  ES=1180  SS=1180  CS=1180  IP=0100   NV UP DI PL NZ NA PO NC ┘内容表示
1180:0100 BB8000          MOV     BX,0080
-RCX
CX 0010  ┐
:0000    ┘CXレジスタの内容を0000に変更している
-RF                          ┌──ゼロフラグとキャリーフラグをセットした
NV UP DI PL NZ NA PO NC -ZR CY   ※セットするフラグの並べ方の順序は必要なものを任意の順でよい
```

## ● S（Search／サーチ）

【書式】 Sレンジ，リスト

指定された範囲で指定されたリストと同じデータを探し，そのアドレスを表示します．

【例】

```
-DCS:100,11F
1180:0100  BB 80 00 B9 80 00 C6 07-00 43 E2 FA B4 4C CD 21   ;..9..F..Cbz4LM!
1180:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-SCS:100,11F CD,21  ←CSの100～11FのメモリにCD，21というデータがあるかサーチを行う
1180:010E ←──────── このアドレスに目的のデータがあったことを表す
```

## ● T（Trace／トレース）

【書式】 T[＝アドレス][,値]

アドレスで示される番地から値で示されるステップ数だけトレースを行いま

す．アドレスが省略されると現在のIP，値が省略されると1とみなされます．トレースとは，プログラムをステップごとに実行しながら各ステップのレジスタ内容およびそのステップの命令を表示することです．

【例】

```
-T=100  ←────CS：100から1ステップ分をトレース

AX=0000  BX=0080  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1180  ES=1180  SS=1180  CS=1180  IP=0103   NV UP DI PL ZR NA PO CY
1180:0103 B98000        MOV     CX,0080
-T  ←──────────現在のIPから1ステップ分をトレース

AX=0000  BX=0080  CX=0080  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1180  ES=1180  SS=1180  CS=1180  IP=0106   NV UP DI PL ZR NA PO CY
1180:0106 C60700        MOV     BYTE PTR [BX],00                DS:0080=14
```

● U（Unassemble／逆アセンブル）

【書式】　U[レンジ]

レンジで示される範囲のコードを逆アセンブルして表示します．レンジの最終アドレスが省略されると20Hバイト分が逆アセンブルされます．

【例】

－U0,15…CS：0～15Hまでを逆アセンブルする

● W（Write／ファイルへの書き込み）

【書式】W[アドレス [，ドライブ，開始レコード，セクタ数]]

NコマンドでFCBにセットされているファイル名でデータをディスク上に書き込みます．アドレスを省略すると，CS：100からのデータをセーブします．

パラメータなしでWコマンドを行うときは，書き込むデータのバイト数がBX：CXレジスタにセットされていなければなりません．ファイルをロードして修正した後，ファイルの長さが変わらなければ，ファイル名，長さ，スタート・アドレスは正しくセットされたままなので，パラメータなしのWコマンドを与えるだけで済みます．

【例】

－W…CS：100以後のBX：CXで示されるバイト数のデータをFCBにセットされている名前でディスクにセーブします

# 第7章

# MS-DOSテクニカル情報

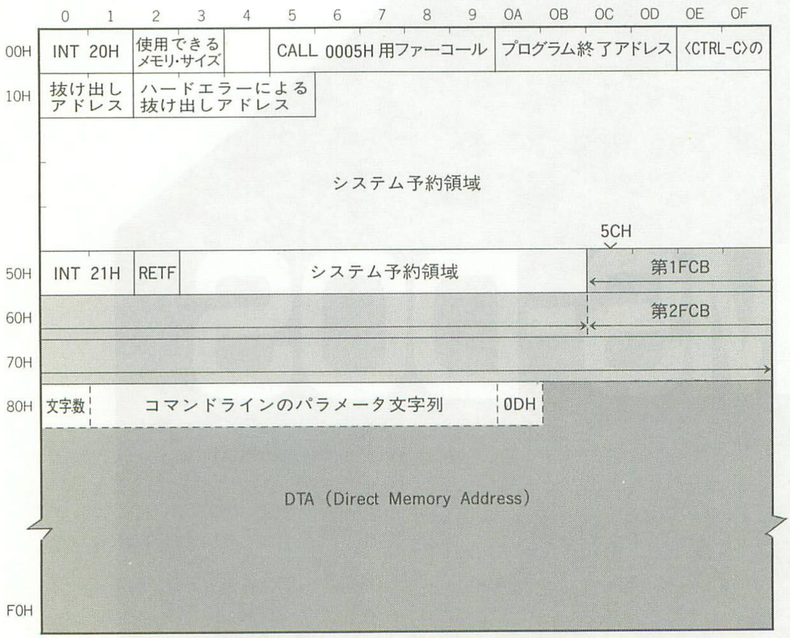
MS-DOS

## 本章の目的

MS-DOS上でファイル・アクセス関係のプログラムを作るときに必要となるPSP，FCB，DTA，ディレクトリ，ファイル・ハンドルなどの技術情報をここでまとめて解説します．

本章は，第8章，9章で学ぶ内容に関する資料として必要なので，それらを読む前に一読しておいてください．

▼図7.1　PSP(プログラム・セグメント・プリフィクス)の構成(説明は次のページ)



# 7.1／PSP(プログラム・セグメント・プリフィクス)

実行可能ファイル（.COMや.EXE）がメモリの空き領域へロードされたとき，その領域を**プログラム・セグメント**と呼びます．MS-DOSはこのプログラム・セグメントの前（プリフィクス）に100Hバイトの作業領域を確保し，そこにコマンドラインからのパラメータや各種エントリ・ポイントを格納しておきます．さらにファイルをアクセスするためのFCBやDTA，その他の作業領域としてもMS-DOSシステムが使用することになります．これらのために使用される領域を**PSP（プログラム・セグメント・プリフィクス）**と呼びます．PSPの構成を図7.1に示します．

PSPの領域は次のように割り当てられています．

00H〜01H …プログラムを終了するためのINT 20Hが置かれる

02H〜03H …使用できるメモリのサイズ（最上位のセグメント・アドレス）が入っている

05H〜09H …CP／M-80のシステムコール(CALL 0005H）との互換性を考えてCALL 0005H用のシステムコールへのエントリが置かれる

0AH〜0DH …プログラム終了アドレス（IP，CS）

0EH〜11H …<CTRL－C>の抜け出しアドレス（IP，CS）

12H〜15H …ハード・エラーによる抜け出しアドレス（IP，CS）

50H〜53H …INT 21H，RETFが置かれる．システムコールのためにここをFAR CALLしても構わないが，あまり使わない

5CH〜7FH …FCB（File Control Block）領域．コマンドラインのパラメータの第1引数が第1 FCBに，第2引数が第2 FCBに格納される．一般に第1 FCBがオープンされると，この領域は次の本章7.2で示すように割り振られる

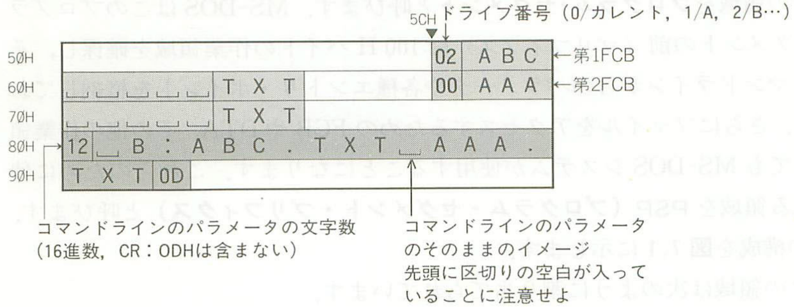
80H〜FFH …DTA（Disk Transfer Address）領域．ディスクとのデータのリード／ライト用のバッファ．最初コマンドラインのパラメータがそのままのイメージで格納されており，80Hにはパラメータの文字列の文字数が入っている．

実際にコマンドラインからパラメータが入力された場合を示します．今，次のような入力がされたとします．

A＞TEST ␣ B：ABC．TXT ␣ AAA．TXT ↵

↑ 第1引数（B：ABC．TXT）　↑ 第2引数（AAA．TXT）

このときPSPの5CH以後には，次のような文字列が格納されることになります．




# 7.2 FCBとDTA

ディスクとのリード／ライトを行うときには，FCB（File Control Block）とDTA（Disk Transfer Address）という領域を使用します．

FCBとDTAのディフォルト領域はPSP内に確保されますが，ユーザが独自の領域に設定することもできます．DTAの変更はファンクション1AHにより行われ，FCBの通知はファイルのオープン時にFCBの先頭アドレスを渡すことによって行われます．

FCBとDTAを用いたファイルのリード／ライトの概要を図7.2に示します．

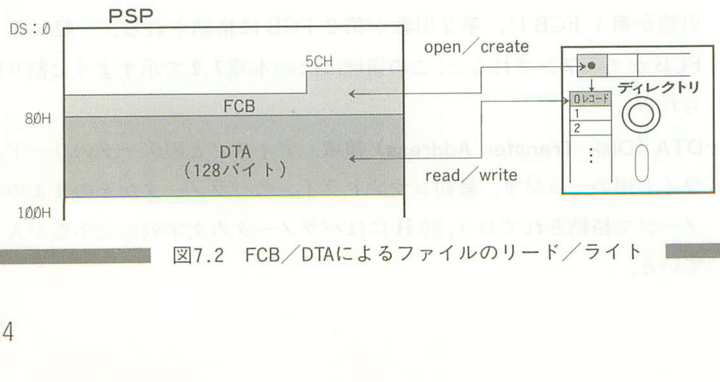



図7.2 FCB／DTAによるファイルのリード／ライト

FCBへのファイル名（d：filename.typ）をセットする方法には2通りあります．1つはプログラム中から直接該当する領域（5CH～）にファイル名をセットする方法で，もう1つはプログラム実行時にコマンドラインのパラメータをFCBへ自動的に渡す方法です（**図7.3**参照）．




図7.3　コマンドラインからパラメータをFCBに渡す

どちらの方法で行っても，FCBにセットされたファイル名でファイルをオープンすると，FCBは**図7.4**に示すような各種情報フィールドに割り当てられます．




| | |
|---|---|
| d | ドライブ番号（0／カレント・ドライブ，1／ドライブA，2／ドライブB…） |
| filename | ファイル名（8文字以内） |
| typ | ファイルのタイプ（3文字以内） |
| cb | 現在のブロック番号（ファイル・オープンにより0にセットされる） |
| rs | 論理レコード長（ファイル・オープンにより0にセットされる） |
| fs | ファイルのサイズ（バイト） |
| dt | 最終更新時刻 |
| dm | 最終更新日付 |
| cr | 現在のレコード番号（ユーザが初期設定する） |
| rr | ランダムレコード番号（rsが64以上ならばCP／Mと同じく3バイト使用） |

図7.4　FCBの構成

シーケンシャル・ファイルの位置はカレント・ブロック番号cbとカレント・レコード番号crによって管理されます．このようすを**図7.5**に示します．

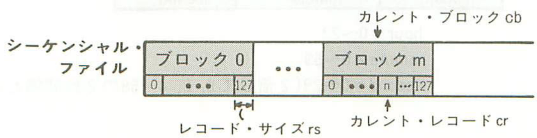



図7.5　カレント・ブロックとカレント・レコード

1ブロックは常に128レコードで構成されます．レコードのサイズはディフォルト値が128バイトになっていますが，1〜65535の範囲で任意に変えることができます．ファイルのオープン時にはcb＝0，cr＝0に初期設定され，これを指定し直さない限りファイルの先頭からリード／ライトが行われます．cr，cbはリード／ライトが行われる毎に自動的に更新されていきます．

ランダム・ファイルの位置はランダム・レコード番号により管理されています．ランダム・レコード番号はファイルの先頭を0として，そこからファイル・エンドまで通し番号を付けたものです．

1レコードのサイズはrsで決められています（図7.6参照）

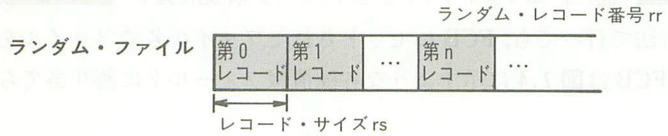



図7.6 ランダム・レコード

ファイル・サイズrsが64以上ならば，rsフィールドは3バイトで済みますが，rsが64以下の場合は4バイト必要になります．

FCBのdtにはファイルの作成時刻，dmにはファイル作成日付がセットされます．そのフィールドのフォーマットは図7.7のようになっています．

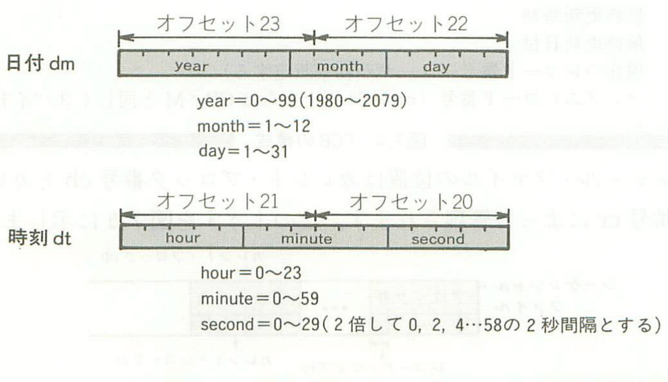



図7.7 日付と時刻のフィールド

# 7.3／フロッピィディスク・レイアウト

MS-DOSのフロッピィディスク・レイアウトの一例を図7.8,7.9に示します．

| トラック＼サイド・セクタ | 0 (1 2 3 4 5 6 7 8) | 1 (1 2 3 4 5 6 7 8) |
|---|---|---|
| 0 | IPL / FAT 1 / FATのコピー 3 / ディレクトリ 5 | ディレクトリ 8 / B データ F |
| 1 | 10 データ 17 | 18 データ 1F |
| 2 | 20 27 | 28 2F |
| ⋮ | | |
| 76 | 4C0 データ 4C7 | 4C8 データ 4CF |

| | |
|---|---|
| 1物理トラック当たりのセクタ数 | 8 |
| サーフェス（サイド）数 | 2 |
| セクタのサイズ（バイト） | 1024 |
| ディレクトリ数 | 192 |
| ディスク容量（キロバイト） | 1232 |
| 相対セクタ番号 | 0〜4CF |

図7.8　MS-DOS 1Mタイプ(5″2HD, 8″2D)フロッピィディスク・レイアウト

| トラック＼サイド・セクタ | 0 (1 2 3 4 5 6 7 8) | 1 (1 2 3 4 5 6 7 8) |
|---|---|---|
| 0 | IPL 0 / FAT 1 / FATのコピー 3 / ディレクトリ 5 | ディレクトリ 8 / C データ F |
| 1 | 10 データ 17 | 18 データ 1F |
| 2 | 20 27 | 28 2F |
| ⋮ | ⋮ | ⋮ |
| 79 | 4F0 データ 4F7 | 4F8 データ 4FF |

| | |
|---|---|
| 1物理トラック当たりのセクタ数 | 8 |
| サーフェス（サイド）数 | 2 |
| セクタのサイズ（バイト） | 512 |
| ディレクトリ数 | 112 |
| ディスク容量（キロバイト） | 640 |
| 相対セクタ番号 | 0〜4FF |

図7.9　MS-DOS 640KBタイプ(5″2DD)フロッピィディスク・レイアウト

フロッピィディスクのデータは**トラック番号**，**サーフェス（サイド）番号**，**セクタ番号**さえ知ることができれば，見つけ出すことができます．8"2Dのフロッピィディスクでは，トラックが0～76の77本，1論理トラック(サーフェス0，サーフェス1を合わせたもの)のセクタ数は16セクタからなります．1セクタのサイズは1024バイトです．

しかしMS-DOSでは，**相対セクタ番号**という通し番号で管理しているので，このような物理的なトラック番号，サーフェス番号，セクタ番号を用いないで，フロッピィディスクのデータをアクセスすることができます．

相対セクタ番号はトラック0，サーフェス0，セクタ1の位置を0にし，以下最後のセクタまで通し番号を付けたものです．先の8"2Dのフロッピィディスクの例では，0～4CFHの相対セクタ番号が付けられています．

相対セクタ番号を用いたレイアウトでは，トラック0には**IPL***，**FAT***，**ディレクトリ***など，一般にはDIRコマンドによって内容を表示することのできないデータが入っています．データの先頭の相対セクタ番号は0BH（5"2DDでは0CH）からです．

●表7.1　フロッピィ・ディスク諸元（MS-DOSユーザーズマニュアルより引用）

| 媒体 ＼ 諸元 | 8インチ | | 5インチ | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 片面単密 | 両面倍密 | 片面倍密 | | 両面倍密 | | 両面倍密倍トラック | 両面高密 |
| 媒体容量(キロバイト) | 250.25 | 1232 | 160 | 180 | 320 | 360 | 640 | 1232 |
| ディスク容量(キロバイト) | 246.5 | 1221 | 156.5 | 175.5 | 315 | 354 | 634 | 1221 |
| トラック数 | 77 | 154 | 40 | 40 | 80 | 80 | 160 | 154 |
| セクタ数/物理トラック | 26 | 8 | 8 | 9 | 8 | 9 | 8 | 8 |
| セクタサイズ（バイト） | 128 | 1024 | 512 | 512 | 512 | 512 | 512 | 1024 |
| ディレクトリ数（ルート） | 68 | 192 | 64 | 64 | 112 | 112 | 112 | 192 |
| 予約セクタ数 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| FAT数 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| セクタ数／FAT | 6 | 2 | 1 | 2 | 1 | 2 | 2 | 2 |
| セクタ数/クラスタ | 4 | 1 | 1 | 1 | 2 | 2 | 2 | 1 |
| FAT ID | FE | FE | FE | FC | FF | FD | FB | FE |

＊NOTE

● IPL　…Initial Program Loader．MS-DOSシステムの立ち上げ用．

● FAT　…File Allocation Table．ファイルやサブ・ディレクトリがディスクのどこのセクタを使用しているかを知るための情報が格納されているテーブル．重要なテーブルなので相対セクタ3，4にそのコピーが格納されている．

●ディレクトリ…ファイル名やディレクトリ名と，それらのファイルに関する情報(属性，作成日時，クラスタ番号，ファイルのサイズ)が格納されている．1ディレクトリで32バイト占有する．

# 7.4／ディレクトリ

MS-DOSのディレクトリは32バイト単位で1つのファイルを管理するようになっています．ディレクトリの構造を**図7.10**に示します．

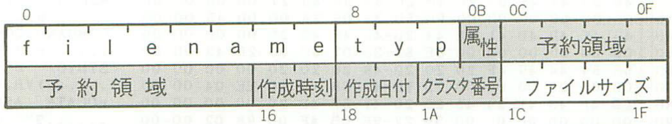



図7.10　ディレクトリの構造

ファイルの属性は**図7.11**のように各ビットをセット／リセットすることによって決まります．

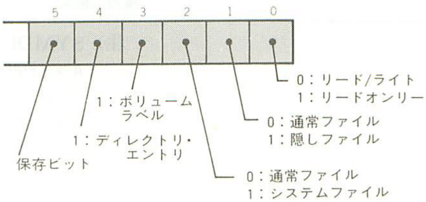



図7.11　ファイルの属性

ファイル作成時刻／作成日付のフォーマットは**図7.7**と同じです．

クラスタ番号には，ファイルやディレクトリの開始クラスタ番号が入っています．これを元にしてFAT内のクラスタ番号をリスト状に検索して，ファイルのデータをアクセスすることができます．クラスタ番号については本章**7.5**を参照してください．

ファイルのサイズはオフセット1FHが最上位バイト，オフセット1CHが最下位バイトになっています．

ディレクトリの実際をDEBUGを用いてダンプした具体例を220ページの**図7.12**に示します．ディレクトリの先頭バイトがE5または00ならば，このディレクトリは未使用であることを示します．もう少し詳しく言うと，フォーマット時には，すべてのディレクトリの先頭に00が書き込まれ，ディレクトリは先頭から空いているところに順序よく取られていくので，先頭に00が置かれているディレクトリの後には，使用中のディレクトリはないと判断することがで

きます．また，一度作成したファイルをDELコマンドなどで消去すると当該ディレクトリの先頭バイトにE5が書き込まれます．

▼図7.12　ディレクトリの例

1度も使われていないディレクトリ
DELにより消去されたファイルを示すディレクトリ
属性(システム/隠し/リードオンリー)
開始クラスタ番号
ファイルサイズ

```
A>DEBUG
-LDS:0,1,5,1
-D0,FF
116C:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00   IO      SYS'....
116C:0010  00 00 00 00 00 00 0B 83-D9 0A 02 00 00 A0 00 00   .........Y......
116C:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00   MSDOS   SYS'....
116C:0030  00 00 00 00 00 00 F0 5D-36 09 2A 00 00 45 00 00   ......p]6.*..E..
116C:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00   COMMAND COM ....
116C:0050  00 00 00 00 00 00 6E 82-33 07 3C 00 26 43 00 00   ......n.3.<.&C..
116C:0060  53 54 44 49 4F 20 20 20-48 20 20 20 00 00 00 00   STDIO   H   ....
116C:0070  00 00 00 00 00 00 7D 56-42 0B 4D 00 EE 04 00 00   ......}VB.M.n...
116C:0080  E5 4F 43 41 54 45 20 20-41 53 4D 20 00 00 00 00   eOCATE  ASM ....
116C:0090  00 00 00 00 00 00 39 22-3E 0B 4F 00 98 02 00 00   ......9">.O.....
116C:00A0  45 58 31 20 20 20 20 20-41 53 4D 20 00 00 00 00   EX1     ASM ....
116C:00B0  00 00 00 00 00 00 43 33-01 0B 50 00 E8 01 00 00   ......C3..P.h...
116C:00C0  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5   .eeeeeeeeeeeeeee
116C:00D0  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5   eeeeeeeeeeeeeeee
116C:00E0  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5   .eeeeeeeeeeeeeee
116C:00F0  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5   eeeeeeeeeeeeeeee
```

属性(通常ファイルでリード/ライト)

注）SYMDEBは次のように入力する．
A＞SYMDEB
－LDS：0␣1␣5␣1
－D␣0, FF

## 7.5 / FAT (File Allocation Table)

MS-DOSでファイルをアクセスする方法は，ハードウェア側に寄るか，ソフトウェア側に寄るかで，いくつかのレベルに分けられます．

最もハードウェア寄りの指定は，トラック／サーフェス／物理セクタを指定する方法で，次は本章11．3で示した相対セクタ番号で指定する方法です．ダイレクト・ディスク・リード／ライトやDEBUGでのファイルのアクセスには相対レコード番号を指定しています．

さらにソフトウェア寄りのレベルはクラスタによる指定です．システムコールを用いたファイルのアクセスでは，セクタというハードウェアに依存する単位を扱う必要はありませんが，これはDOS（ディスク・オペーティングシステム）がこれらすべてを管理してくれているためです．DOSはファイルをアクセスするときの基本単位をクラスタで管理しています．1クラスタは物理セクタの2のべき乗となるように取られ，一般に1クラスタを1または2セクタとす

る媒体が多いようです．

このファイルをアクセスする際に，どのファイルがどのクラスタを使用しているかという情報が必要になります．**FAT（File Allocation Table）**にはファイルが使用しているクラスタの情報がリスト構造で書き込まれています．ディレクトリのオフセット1AH，1BHのクラスタ番号はFATへのポインタとなっています．このようすを**図7.13**に示します．

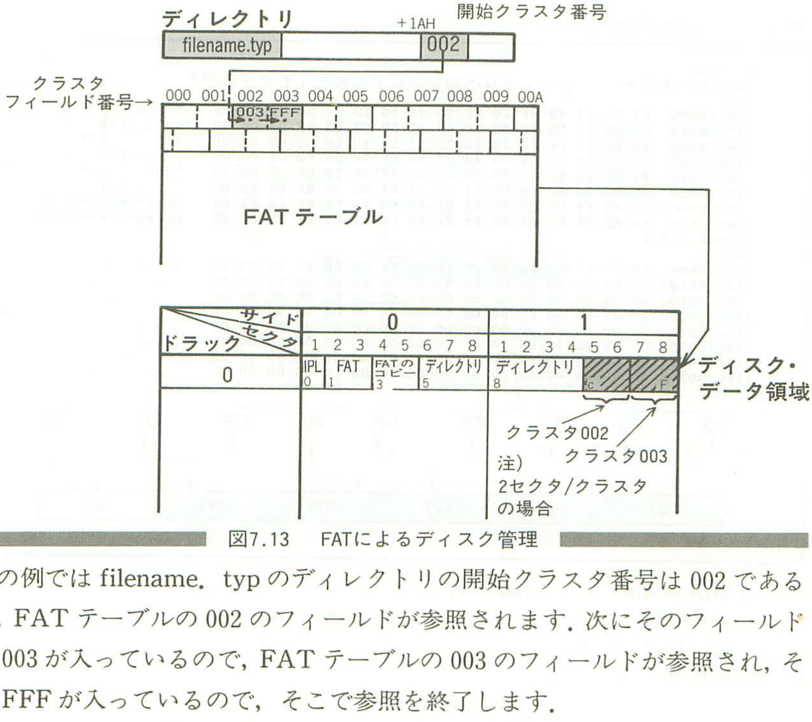



図7.13　FATによるディスク管理

この例ではfilename．typのディレクトリの開始クラスタ番号は002であるから，FATテーブルの002のフィールドが参照されます．次にそのフィールドには003が入っているので，FATテーブルの003のフィールドが参照され，そこにFFFが入っているので，そこで参照を終了します．

このようにして参照したクラスタ番号（この例では002と003）にファイルのデータが入っていることになります．**図7.13**の例では2セクタ／クラスタなので，002のクラスタは相対セクタ番号0CHと0DH，003のクラスタは0EHと0FHに相当します．

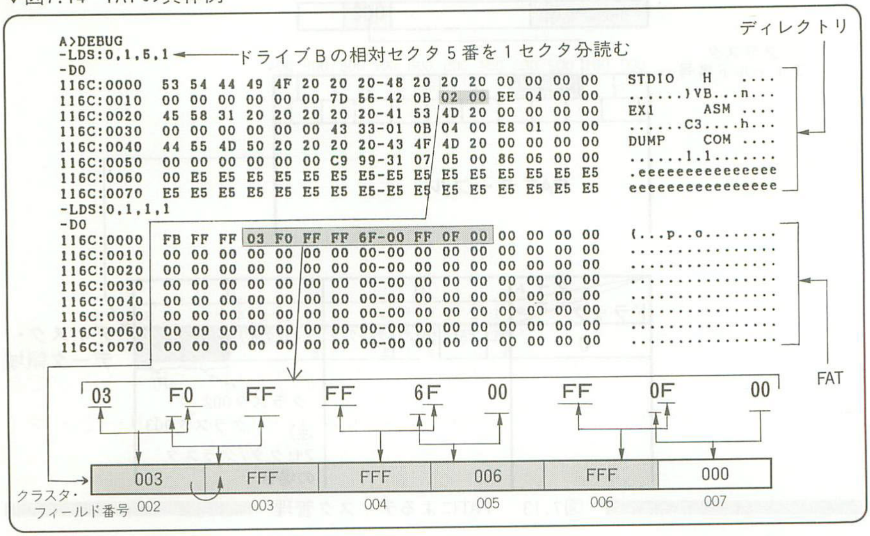
また，1つのファイルのクラスタは連続しているとは限らず，物理的に離れていても1つのファイルとして扱われます．

FATテーブルは3バイトで2つのクラスタ番号を表示しているので，上位・

下位バイトの組み合わせが多少複雑になります．また，FATの先頭1バイトはディスクの種別を示すFAT IDが置かれるため，先頭3バイトを飛ばして4バイト目からFATテーブルとして使用します．したがってクラスタ番号の000と001は未使用で，002がクラスタ番号の最初の番号となります．

FATの具体例を図7.14に示します．

▼図7.14 FATの具体例




注）SYMDEBは次のように入力する．

A＞SYMDEB

－LDS：0␣1␣5␣1

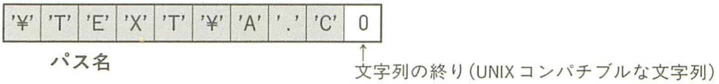
－D␣0

# 7.6／ファイル・ハンドル

FCBとDTAを用いたファイルのアクセスは，CP／Mとの互換性のために設けられているもので，**階層ディレクトリ構造**を扱うことができません．

MS-DOS ver.2.0以降では，階層ディレクトリを扱えるUNIXライクなシステムコールが追加され，ファイルのアクセスには**ファイル・ハンドル**というファイル記述子（ファイル・ディスクリプタ）を用います．

ファイルをオープンするときのファイル名（パス名）は，ファイル名のアスキー文字列の最後に0（ヌル）を置いたものとします．




この文字列の先頭アドレスをファイルをオープンするシステムコール（ファンクション3DH）に渡すとファイル・ハンドル番号が返されます．以後，ファイルへのリード／ライトは，ファイル・ハンドル番号を用いて行うことができます．ユーザ側は何バイトのデータをどこの領域（バッファ）にリード／ライトするのか指示するだけでよく，後はMS-DOSが作業領域を用いてレコード番号やブロック番号を管理してくれます．そのようすを**図7.15**に示します．

MS-DOSの起動時に，0～4のファイル・ハンドル番号は，標準入出力にオープンされているので，ユーザがファイルをオープンすると一般にファイル・ハンドルは5番から割り当てられていきます．

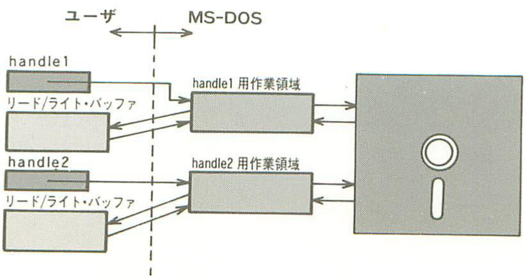



図7.15　ファイル・ハンドルの概念

# 第8章

# MS-DOSシステムコール

# INT 21H

**本章の目的**

アセンブリ言語でプログラムを作成する場合に，コンソール，プリンタ，ディスク・ドライブなどへアクセスするルーチンをプログラマが自前で作るとなると，非常に労力が要ることになります．

一般にOSには，これらの周辺機器へのアクセスを支援するためのサブルーチンが用意されており，このサブルーチンをコールすることを**システムコール（ファンクションコール）**と呼びます．

MASMではこれらのシステムコールをマクロ定義しておくことで，高級言語で行う記述に近いプログラミングができます．そこでここではシステムコールの中から特によく使うコンソール入出力，FCBを介したファイル入出力，ファイル・ハンドラを介したファイル入出力に的を絞ってその使い方について説明します．

# 8.1／システムコールとは

システムコールというものを説明するにあたって，まず，"hello"という文字をCRTスクリーンに表示させる場合を考えてみることにします．

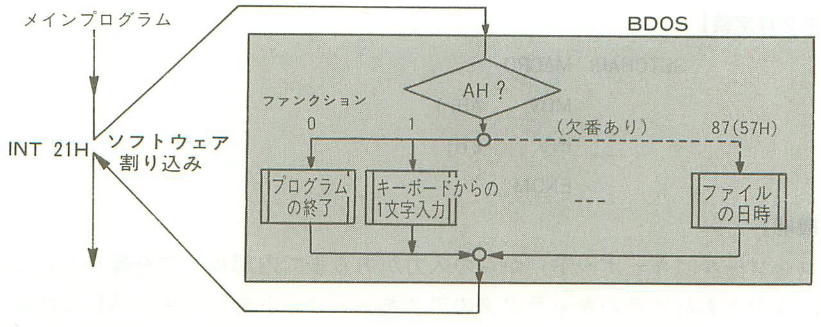
BASICではPRINT "hello"とすれば良いのですが，アセンブリ言語ではそんなに簡単にはいきません．まず，CRTが接続されている**I／Oポート**を捜し，そのポートに**OUT命令**で'h'，'e'，'l'…のアスキー・コードを順次送らなければなりません．

MS-DOSでは（他のOSでも同様ですが）こうしたいくつかの基本的な処理をサブルーチンとして**BDOS（MSDOS．SYS）**内に組み込んでおいて，ユーザがこれをコールすることができるようにしています．これをシステムコールと呼び，MS-DOSには約70種類のサブルーチンが用意されています．

CP／M-80ではシステムコールをCALLで行っていましたが，CP／M-86やMS-DOSでは**INT命令によるソフトウェア割り込み**を用いて行います．このようすを**図8.1**に示します．

MS-DOSのシステムコールは，レジスタAHにファンクションコード0

図8.1　MS-DOSのシステムコールの概念



~87 (57 H) を入れ，必要なパラメータ（ないこともある）をレジスタ CX，DX などに入れて，ソフトウェア割り込みの INT 21 H を行います．コール書式は次の通りです．

```
MOV    AH,ファンクションコード
[DX,   CX  ←  パラメータ]
INT    21H
```

システムコール後のレジスタの内容は，レジスタ AX などのリターン情報が入るもの以外はそのまま保存されます．

巻末に付録として，MS-DOS のシステムコール一覧が載せてあるので参考にしてください．システムコールのより詳しい説明はマニュアルまたは他の解説書を参考にしてください．

## 8.2 コンソール入出力関係のシステムコール

ファンクションコール 1 ~12 (0 CH) まではコンソールおよびプリンタに対して入出力を行うものです．これらは比較的単純なシステムコールです．

このシステムコールは CP／M-80, CP／M-86, MS-DOS の間では，機能的には互換性があるものと考えて構いません．

### ●コンソールからの入力 [function 1 : getchar]

**【マクロ定義】**

```
GETCHAR  MACRO
         MOV    AH, 1
         INT    21H
         ENDM
```

**【機能】**

コンソール（キーボード）からの入力があるまで内部ループを繰り返していて，入力があればそのキャラクタのアスキー・コードをレジスタ AL に格納してリターンします．入力と同時にコンソールにエコーバックされます．

**●コンソールへの１文字出力［function 2 ：putchar］**

**【マクロ定義】**

```
PUTCHAR  MACRO  ASC
         MOV    AH, 2
         MOV    DL, ASC
         INT    21H
         ENDM
```

**【機能】**

レジスタDLにセットされたキャラクタをコンソール(スクリーン)に表示します．

**【例】**

putchar　‘A’　…‘A’がスクリーンに表示される

putchar　CL　…レジスタCLの内容がスクリーンに表示される

**●プリンタへの１文字出力［function 3 ：lptout］**

**【マクロ定義】**

```
LPTOUT  MACRO  ASC
        MOV    AH, 5
        MOV    DL, ASC
        INT    21H
        ENDM
```

**【機能】**

レジスタDLにセットされたキャラクタをプリンタに出力します．

**【例】**

lptout　‘A’　…‘A’がプリンタに出力される

lptout　CL　…レジスタCLの内容がプリンタに表示される

●ダイレクトコンソール入力［function 6：inkey］

【マクロ定義】

```
INKEY   MACRO
        MOV     AH,6
        MOV     DL,0FFH
        INT     21H
        ENDM
```

【機能】

ファンクションコール6は，レジスタDLにセットする値がFFHのときに入力モードになり，それ以外は出力モードとなります．

ファンクション6の入力モードは，コンソール（キーボード）から入力があればそのアスキー・コードをレジスタALに，入力がなければ0（ヌル）をレジスタALに格納してリターンします．

ファンクション1（getchar）のような「入力があるまで内部ルーチンを繰り返す」ということはなく，入力があってもなくてもリターンします．

さらに入力文字をエコーバックせず，<CTRL＋C>の検査も行いません．

●文字列のコンソールへの出力［function 9：print］

【マクロ定義】

```
PRINT   MACRO   MSG
        MOV     AH,9
        MOV     DX,OFFSET   MSG
        INT     21H
        ENDM
```

【機能】

文字列が格納されている先頭アドレス（オフセット）をレジスタDXに格納し，コールすると文字列がスクリーンに表示されます．文字列の最後には必ず目印として‘$’を置かなければなりません．

```
MOV   DX,OFFSET   MSG
```

の代わりに，

```
LEA　DX, MSG
```

としても構いません．

【例】

```
          ⋮
          ⋮
          print　MSG 1
          ⋮
          ⋮                cr       lf      文字列の最後の印
                           ↓        ↓       ↓
MSG 1   DB      'Hello', 0 DH, 0 AH, '$'
```

●コンソールからの文字列入力［function　0 AH（10）：input］

【マクロ定義】

```
INPUT   MACRO   CONBUF
        MOV     AH, 0AH
        MOV     DX, OFFSET   CONBUF
        INT     21 H
        [putchar 0AH]
        ENDM
```

【機能】

コンソール（キーボード）から入力された文字列を，レジスタ DX で示されるアドレス(オフセット値)＋2 番地の領域に順次格納します．入力の終わりは ⏎ キーで，それ以前であれば入力文字の修正をすることができます．

入力できる最大文字数は，最後のリターンコード(0 DH)を含めて 255 (FFH) です．入力バッファの構成を**図 8.2** に示します．

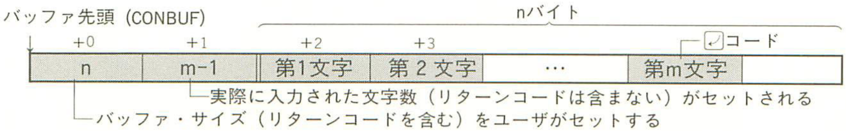



図8.2　入力バッファの構成

バッファの先頭には，入力できる最大文字数をあらかじめユーザがセットしておかなければなりません．キー入力がこの値を超えると警告音が発せられ，それ以上は入力できなくなります．バッファの先頭＋1番地には，実際に入力された文字数（↵キーのコードは含まない）がセットされます．

一般に入力の終わりの↵キーでカーソルはその行の先頭に戻りますが，改行（LF）はしません．そこでマクロ input の最後で，

```
putchar    0AH
```

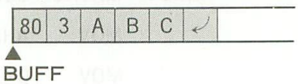
を加えておけば，↵キーの入力で CR・LF（復帰・改行）を行うことができます．

また，入力バッファ（CONBUF）の先頭へ入力文字数をセットする部分もマクロ input の中に含めることもできます．

**【例】**

```
            MOV     BUFF, 80
            input   BUFF
                      :
                      :
BUFF    DB          82 DUP （？）
```

●コンソールから A B C ↵ と入力したときのバッファ BUFF は次のようになる．



# 8.3／スタンダードI/Oヘッダの作成

前章**8.2**で説明したいくつかのコンソール入出力関係およびセグメントの定義に関するマクロを stdio.h というマクロ定義ファイルとして作成します．これをユーザプログラムの先頭で，

```
INCLUDE   STDIO.H
```

と INCLUDE 擬似命令でヘッダ・ファイルとして取り込みます．

stdio.h のリストを**リスト 8.1** に示します．リスト中のマクロで，前章 8.2 で説明してないものについてここで説明します．

● pseg

コード・セグメントの宣言

● pend

コード・セグメントの終わり

● dseg

データ・セグメントの宣言

● dend

データ・セグメントの終わり

● sseg

スタック・セグメントの宣言と領域の確保

● retms

プログラムの終了と MS-DOS への復帰

● lpbuf

マクロ input で入力したバッファ内データをプリンタに出力するものです．文字列データの先頭は BUF＋2，データの終わりの印は 0 DH です．0 DH を受信すると，0 AH（LF）を送り改行を行います．



● lprint

マクロ print のプリンタへの出力版です．データの先頭は MSG，データの終わりの印は‘$’です．

```
MSG
↓
| 文 字 列                              | $ |
```

●リスト8.1　stdio.h

```
; --------------------------------------
; *   console I/O macro library        *
; --------------------------------------

ESC     EQU     1BH
SPACE   EQU     20H
CR      EQU     0DH
LF      EQU     0AH

PSEG    MACRO                   ; program segment
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA,SS:STACK
        ENDM

PEND    MACRO
CODE    ENDS
        ENDM

DSEG    MACRO                   ; data segment
DATA    SEGMENT
        ENDM

DEND    MACRO
DATA    ENDS
        ENDM

SSEG    MACRO                   ; stack segment
STACK   SEGMENT STACK
        DW      128 DUP(?)
STACK   ENDS
        ENDM

RETMS   MACRO                   ; return MS-DOS
        MOV     AH,4CH
        INT     21H
        ENDM

GETCHAR MACRO
        MOV     AH,1
        INT     21H
        ENDM
```

```
PUTCHAR MACRO   ASC
        MOV     AH,2
        MOV     DL,ASC
        INT     21H
        ENDM

LPTOUT  MACRO   ASC       ; putchar for printer
        MOV     AH,5
        MOV     DL,ASC
        INT     21H
        ENDM

INKEY   MACRO
        MOV     AH,6
        MOV     DL,0FFH
        INT     21H
        ENDM

PRINT   MACRO   MSG
        MOV     AH,9
        MOV     DX,OFFSET MSG
        INT     21H
        ENDM

INPUT   MACRO   CONBUF
        MOV     AH,10
        MOV     DX,OFFSET CONBUF
        INT     21H
        PUTCHAR 0AH
        ENDM

LPBUF   MACRO   BUF       ; input buffer printer out
        LOCAL   START,FIN
        MOV     BX,OFFSET BUF+2
START:  LPTOUT  [BX]
        CMP     DL,CR
        JE      FIN
        INC     BX
        LOOP    START
FIN:    LPTOUT  LF
        ENDM

LPRINT  MACRO   MSG
        LOCAL   START,FIN
        MOV     BX,OFFSET MSG
START:  MOV     AH,5
        MOV     DL,[BX]
        CMP     DL,'$'
        JE      FIN
        INT     21H
        INC     BX
        JMP     START
FIN:
        ENDM
```

# 8.4 stdio.hファイルの利用例

stdio .hファイルを取り込むことにより，ユーザ・プログラムの負担は軽くなり，高級言語の記述に近い表現が可能となります．ここではstdio .hファイルを用いた簡単なプログラムを例示することにします．

なおマクロコール部は，それがマクロであることがはっきりするように小文字で記述しましたが，当然大文字で記述しても構いません．

## 入力した文字でスクリーンを埋める

**プログラム**

```
; ----------------------------
; *        crt fill          *
; ----------------------------

INCLUDE B:STDIO.H

        sseg            ; stack segment

CODE    SEGMENT
        ASSUME  CS:CODE,SS:STACK
START:  getchar
        MOV     CX,80*23-1
FIL:    putchar AL
        LOOP    FIL
        retms
CODE    ENDS
        END     START
```

マクロgetcharでレジスタALに入力された文字を得て，マクロputcharでスクリーンに出力します．

スクリーンを埋めるためには，横80文字，縦23行を埋めればよいので，ループ回数は80 ＊ 23－1となります．－1しているのは，既に文字を入力した際に1文字分がエコーバックされているための処置です．

### ミニ・タイプライタ

**プログラム**

```
; -------------------------------
; *        mini typewriter        *
; -------------------------------

INCLUDE B:STDIO.H

        dseg                ; data segment
MSG     DB      'Please Key in !',CR,LF,'$'
CONBUF  DB      80
        DB      81 DUP(?)
        dend

        sseg                ; stack segment

        pseg                ; program segment
START:  MOV     AX,DATA
        MOV     DS,AX
        print   MSG
NEXT:   input   CONBUF
        lpbuf   CONBUF
        JMP     NEXT
        retms
        pend
        END     START
```

キーボードから入力した文字をプリンタに出力するミニ・タイプライタを作成します．

文字列の入力はマクロ input によりバッファ CONBUF に行われます．これをマクロ lpbuf によりプリンタに出力します．

バッファ CONBUF の先頭には，DB 擬似命令であらかじめ入力する最大文字数（この例では 80）をセットしておきます．

### ダイレクト・コンソール入力

**プログラム**

```
; -------------------------------
; *    direct console input      *
; -------------------------------

INCLUDE B:STDIO.H

        sseg                ; stack segment

CODE    SEGMENT
        ASSUME  CS:CODE,SS:STACK
```

```
START:   MOV      CL,')'
NEXT:    inkey
         CMP      AL,0
         JE       BRN
         MOV      CL,AL
BRN:     putchar  CL
         JMP      NEXT
CODE     ENDS
         END      START
```

ダイレクト・コンソール入力を行うマクロ inkey はキー入力があってもなくてもリターンして来るので，getchar のようにそこでプログラムが停止してしまうことはありません．

このプログラムは，マクロ inkey で入力された文字（ヌル以外）をスクリーンに表示し続け，その間たえず inkey でキー入力を検知しています．新たに inkey に文字の入力があれば出力キャラクタをその入力キャラクタに変更して，また表示し続けます．

## 8.5 FCBを介したファイルの入出力

ファイル入出力関係のシステムコールは数多くありますが，ここではシーケンシャル・ファイルのリード／ライトをする上で，基本的に必要なものだけについて説明します．

ファイル入出力を行う上で，FCB（File Control Block）とDTA（Disk Transfer Address）についての知識が必要であるので，必要であるならば，前章7.2を参考にしてください．

●ファイルのオープン［function 0FH（15）：open］

【マクロ定義】

```
OPEN    MACRO   FCB
        MOV     AH, 0FH
        MOV     DX, OFFSET  FCB
        INT     21H
        ENDM
```

【機能】

レジスタDXで示されるFCBアドレス(DSセグメント内のオフセット値)にセットされているファイル名のファイルをディスクから探し出し，そのディレクトリの情報をFCBにコピーします.

この一連の処理が終わった状態をファイルがオープンしたといい，以後このファイルに対して読み取り・書き込み(リード／ライト)を行うことができます.

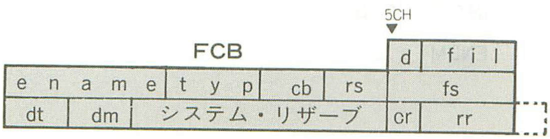
ファイルが見つからなかった場合には，レジスタALにFFHがセットされます.ファイルが見つかれば，レジスタALに0をセットし，FCBの各フィールドを次のように設定します.

①FCBにセットされているドライブdの値が00（カレント・ディスク）ならば，具体的なドライブ番号（01＝A，02＝B…）に替えます.

②カレント・ブロック番号の入るフィールドcbを0にセットします.

③レコード・サイズの入るフィールドrsを128（ディフォルト値）にセットします.

④ファイル・サイズの入るフィールドfs, 日時の入るフィールドdm／dtにディレクトリの内容がコピーされます.




●シーケンシャル・ファイルの読み取り［function　14H（20）：read］

【マクロ定義】

```
READ    MACRO   FCB
        MOV     AH, 14H
        MOV     DX, OFFSET  FCB
        INT     21H
        ENDM
```

【機能】

FCB内のカレント・レコード・フィールドcr（ディフォルト7CH番地）が示すシーケンシャル・ファイルの該当レコードをディスクから読み出し，DTA

（ディフォルト 80 H 番地〜FFH 番地）に格納します.

ファイルの先頭から読むには, cr を 0 に初期設定します. 1レコード（rs で決められるが, ディフォルトは 128 バイト）読み取るたびに cr の内容は自動的に＋1されます. レジスタ AL に返されるリターンコードは次のような意味を持ちます.

0：読み取りが正常に行われた

1：ファイルの終わり

2：DTA に 1 レコード読み込むだけの領域がない

3：最後のレコードが読み取られ, 1 レコードに満たない部分を 0 で埋めた

●ファイルの作成［function 16 H（22）：create］

【マクロ定義】

```
CREATE  MACRO  FCB
        MOV    AH, 16 H
        MOV    DX, OFFSET  FCB
        INT    21 H
        ENDM
```

【機能】

FCB にセットされているファイル名でディスク上に新しいディレクトリを開設します. 同一ファイル名のファイルが既にディスク上に存在する場合は, そのファイルを消去し, 新規の空ファイルを作成します.

CP／M-80 や CP／M-86 では, このファイル消去を行わないので, ユーザがファイル作成以前にファイルの消去を行わなければなりません. レジスタ AL に返されるリターンコードは次のような意味を持ちます.

0：正常にファイルが作成された

FF：ファイルが作成できなかった

●シーケンシャル・ファイルへの書き込み［function 15 H（21）：write］

【マクロ定義】

```
WRITE   MACRO   FCB
        MOV     AH, 15 H
        MOV     DX, OFFSET   FCB
        INT     21 H
        ENDM
```

【機能】

DTAのデータ（ディフォルト80 H～FFHの128バイト）をカレント・ブロックcb，カレント・レコードcrによって示されるディスクのファイルにライトします．このライトが行われるとcrの内容は自動的に＋1されます．レジスタALに返されるリターンコードは次のような意味を持ちます．

0：正常にファイルへ書き込まれた

1：ディスクが一杯である

2：DTAに1レコード分の空きがない

●ファイルのクローズ［function　10 H（16）：close］

【マクロ定義】

```
CLOSE   MACRO   FCB
        MOV     AH, 10 H
        MOV     DX, OFFSET   FCB
        INT     21 H
        ENDM
```

【機能】

ファイルの生成(create)により，ディスク上に該当ファイルのディレクトリを開設しますが，1レコードずつ書き込む作業時にはこのディレクトリの更新はまだ行われてはおらず，書き込みが終了したときにFCBの内容をディスクのディレクトリにコピーし，新しいファイルを完成させます．以上の処理をファイルのクローズと呼びます．

ファイルのクローズは，書き込みを行ったときのみ必要で，読み取りだけの場合は行わなくても構いません．レジスタALに返されるリターンコードは次

のような意味を持ちます．

0：正常にクローズが行われた

FF：クローズができなかった

## 8.6 ファイル入出力ヘッダの作成

前節8.5で示したFCBを介して行うファイル入出力関係のマクロをまとめてfile.hという**ヘッダ・ファイル（マクロ定義ファイル）**にしたものを**リスト8.2**に示します．

●リスト8.1　file.h（FCBを介したファイル入出力ヘッダ）

```
; -----------------------------------
; *        file macro libraly        *
; -----------------------------------

CREATE  MACRO   FCB                    ┐
        MOV     AH,16H                 │
        MOV     DX,OFFSET FCB          │ ファイルの新規作成
        INT     21H                    │
        ENDM                           ┘

WRITE   MACRO   FCB                    ┐
        MOV     AH,15H                 │
        MOV     DX,OFFSET FCB          │ ファイルの書き込み
        INT     21H                    │
        ENDM                           ┘

CLOSE   MACRO   FCB                    ┐
        MOV     AH,10H                 │
        MOV     DX,OFFSET FCB          │ ファイルのクローズ
        INT     21H                    │
        ENDM                           ┘

OPEN    MACRO   FCB                    ┐
        MOV     AH,0FH                 │
        MOV     DX,OFFSET FCB          │ ファイルのオープン
        INT     21H                    │
        ENDM                           ┘

READ    MACRO   FCB                    ┐
        MOV     AH,14H                 │
        MOV     DX,OFFSET FCB          │ ファイルの読み取り
        INT     21H                    │
        ENDM                           ┘
```

# 8.7／file.hを用いたシーケンシャル・ファイルのアクセス例

**リスト 8.4** に示した file .h をユーザ・プログラムで

```
INCLUDE   FILE .H
```

として取り込むことにより，ファイルの入出力処理に関するユーザの負担を軽くすることができます．ここでは file .h ファイルを用いた簡単なプログラムを例示することにします．

なお，マクロコール部をそれがマクロであることがはっきりするように，小文字で記述しました．

## シーケンシャル・ファイルの書き込み

**プログラム**

```
; -----------------------------------
; *      sequential file write      *
; -----------------------------------

INCLUDE B:STDIO.H                                     ┐①
INCLUDE B:FILE.H                                      ┘

        dseg                    ; data segment        ┬
        ORG     5CH                                   │
FCB     DB      34 DUP(?)                             │ ②
BUFF    DB      130 DUP(?)                            │
        dend                                          ┴

        sseg                    ; stack segment

        pseg
START:  create  FCB             ; create file         ↕③
        MOV     FCB+32,0        ; cr field set        ↕④
        MOV     BUFF,128        ; input buf size      ↕⑤
NEXT:   MOV     CX,128                                ┬
        MOV     BX,OFFSET BUFF+2                      │
CLEAR:  MOV     BYTE PTR [BX],0 ; buf 0 clear         │ ⑥
        INC     BX                                    │
        LOOP    CLEAR                                 ┴
        input   BUFF            ; data input          ┬
        MOV     AL,BUFF+1                             │ ⑦
        CMP     AL,0            ; input end           │
        JE      FIN                                   ┴
```

```
        write   FCB             ; write         ⑧
        JMP     NEXT
FIN:    close   FCB             ; close file    ⑨
        retms                   ; return ms-dos
        pend
        END     START
```

コマンドラインから与えたファイル名のシーケンシャル・ファイルにコンソールからデータを書き込むプログラムです．このプログラムの基本設計の概念を図 8.3 に示します．

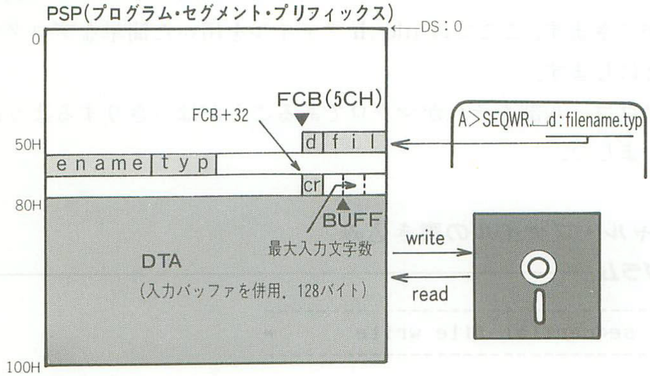



図8.3　シーケンシャル・ファイルのリード／ライトの概念

書き込むファイルのファイル名は次のようにコマンドラインから入力するものとします．

A＞SEQWR　d：filename .typ ↵

これで，FCB に d：filename .typ がセットされます．

function 0 AH（input）で用いるバッファの先頭を DTA－2 番地（7 EH 番地）に設定することにより，入力バッファと DTA バッファを共用します．したがって 1 レコードを 128 バイトのディフォルト値とします．

以下にプログラムの説明を行います．

①ヘッダ・ファイルの取り込み．

②このプログラムでのデータ・セグメントは，FCB と BUFF という変数を用意するためだけのもので，このセグメントへのデータのアクセスは行われない．つまり，プログラムの中で DS をデータ・セグメントの先頭に初期設定しない

ので，DS：0 は PSP の先頭を指したままとなる．したがって FCB，BUFF のオフセットは，PSP 内のオフセット(5 CH と 80 H)を示すことになる．

③ FCB にセットされたファイル名のファイルを新規に生成する．

④ cr（カレント・レコード）フィールドを 0 に初期設定する．

⑤入力バッファ（DTA）の最大入力文字数を 128 バイトにする．

⑥入力バッファ内を 0 クリアする．入力した文字が 128 バイトに満たない場合は残りの文字もゴミとして書き込まれる．

⑦データ入力．「空行 ⏎」のみを入力すると，BUFF＋1 は 0 にセットされるので，これをプログラムの終了と判定することにする．

⑧ DTA の 128 バイトをディスクに書き込む．

⑨ファイルをクローズして MS-DOS に復帰する．

なお，このプログラムでは，一回の入力（128 バイトに満たない場合が多い）を 1 レコードとして書き出しているので，無駄が多く，実用的にするには，バッファが 128 バイト一杯になったら書き出すように変更する必要があります．

### シーケンシャル・ファイルの読み取り

**――プログラム――**

```
; ------------------------------------
; *      sequential file read        *
; ------------------------------------

INCLUDE B:STDIO.H
INCLUDE B:FILE.H

        dseg                        ; data segment
        ORG     5CH
FCB     DB      36 DUP(?)
DTA     DB      128 DUP(?)
        dend

        sseg                        ; stack segment

        pseg
START:  open    FCB                 ; open file          ↕①
        MOV     FCB+32,0            ; cr field set       ↕②
LOOP1:  read    FCB                                      ┬
        CMP     AL,1                ; if file end        │③
        JE      FIN                                      │
        MOV     BX,OFFSET DTA       ; 1 record disp      ┼
LOOP2:  putchar [BX]                                     │④
        INC     BX                                       │
```

```
        CMP      DL,CR           ↑
        JNE      LOOP2           | ④
        putchar  LF              |
        JMP      LOOP1           ↓
FIN:    retms
        pend
        END      START
```

このプログラムは先のシーケンシャル・ファイルの書き込みで作成したファイルを読み込みます．基本的な考えはシーケンシャル・ファイルの書き込みと同じです．以下にプログラムの説明を行います．

①コマンドラインからFCBにセットされたファイルをオープンする

②crフィールドを0に初期設定する．

③1レコード(128バイト)のファイルを読み取る．ファイルの終わりはレジスタALへのリターン・コードが1のときである．

④DTA内に読み込んだデータの最後はCR (0DH：キャリッジ・リターン) なので，このコードが現れるまで1文字ずつスクリーンに表示する．また，改行を行うためにLF (0AH：ライン・フィード) を出力します．

## TYPEコマンドの作成

### プログラム

```
; -------------------------------------
; *         type command             *
; -------------------------------------

INCLUDE B:STDIO.H
INCLUDE B:FILE.H

        dseg                     ; data segment
        ORG      5CH
FCB     DB       36 DUP(?)
DTA     DB       128 DUP(?)
        dend

        sseg                     ; stack segment

        pseg
START:  open     FCB             ; open file       ↕①
        MOV      FCB+32,0        ; cr field set    ↕②
LOOP1:  read     FCB                               ↑
        CMP      AL,1            ; if file end     | ③
        JE       FIN                               ↓
```

```
        MOV     BX,OFFSET DTA   ; 1 record disp
        MOV     CX,128
LOOP2:  MOV     AL,BYTE PTR [BX]
        CMP     AL,1AH          ; file end mark      ④
        JE      FIN
        putchar [BX]
        INC     BX
        LOOP    LOOP2
        JMP     LOOP1
FIN:    retms
        pend
        END     START
```

---

このプログラムはアスキー・ファイルをTYPEするもので，先のシーケンシャル・ファイルの読み取りプログラムに若干手を加えたものです．

以下にプログラムの説明を行います．

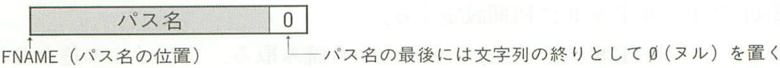
①コマンドラインからFCBにセットされたファイルをオープンする
②crフィールドを0に初期設定する．
③1レコード（128バイト）をファイルから読み取る．ファイルの終わりはレジスタALへのリターン・コードが1のときである．
④先に例示したファイルへの書き込みプログラムでは，1入力で1レコードを書き出しているので1レコード内にはいつも空きが生じていた．一般の高級言語などのファイルではこうした空きが生じないように考慮されている．したがってDTAのデータを128バイト分一杯になるまで表示していけばよい．しかし，最後のレコードは128バイトに満たない場合がある．その場合はファイル・エンド・マークとして，1AHが置かれているので，それを検知したら表示を終了することにする．

一般のテキストエディタで作成したテキストの行末は，CR（0DH），LF（0AH）が置かれているので，LFコードをさらに出力する必要はありません．

## 8.8 ファイル・ハンドラを介したファイルの入出力

本章 **8.6** で説明した FCB によるファイルの入出力は CP／M と互換性を持つために設けられているものですが，この方法では MS-DOSver.2.0 以後の**階層ディレクトリ**には対応できません．これらに対応するためにファイル・ハンドラを介したファイルの入出力用のシステムコールが用意されています（**ファンクションコール 3 CH～46 H，4 EH，4 FH，56 H，57 H**）．

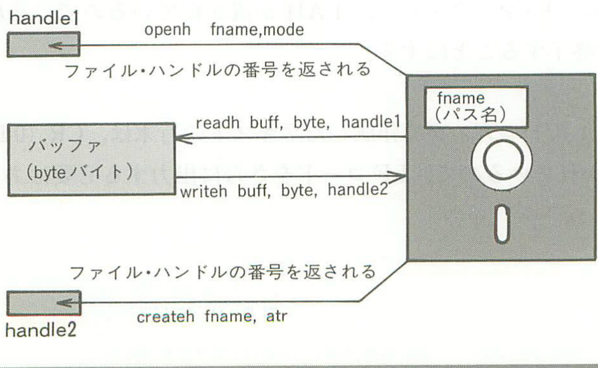
ファイル・ハンドラを介したファイル入出力では，次のようなパス名（パス名を伴ったファイル名）によってファイルが格納されている位置，およびファイルの属性（リードオンリー，リード／ライトなど）を与えて，ファイルを open または create します．




これにより，そのパス名のファイルに対するファイル・ハンドラ番号（ファイル・ディスクリプタ：ファイル識別番号）が返されます．こうしてファイルオープンされたファイルはファイル・ハンドラを用いてリード／ライトすることができます．リード／ライトのバイト数とバッファはユーザが指定します．

以上のようすを**図 8.4** に示します．

図8.4 ファイル・ハンドルによるファイル入出力の概念



●ファイルのオープン［function　3DH（61）：openh］

【マクロ定義】

```
OPENH   MACRO   FNAME, MODE
        MOV     DX, OFFSET  FNAME ; パス名の位置
        MOV     AL, MODE          ; モード
        MOV     AH, 3DH
        INT     21H
        ENDM
```

【機能】

FNAME で示されるパス名のファイルを MODE が示すモードでオープンします．MODE としては次の値が指定できます．

0…ファイルをリードするためにオープンする
1…ファイルをライトするためにオープンする
2…ファイルをリード／ライト用にオープンする

リターン時にキャリーがセットされていなければ，レジスタ AX にファイル・ハンドル番号が返され，ファイルのオープンが正常に行われたことになります．

もし，キャリーがセットされていれば，ファイルのオープンが正常に行われていないことになり，レジスタ AX に次のエラー・コードがセットされます．

AX＝2：ファイルが存在しない
　　4：既にオープンしているファイルの数が多すぎる
　　5：アクセスが否定された
　　12：レジスタ AL にセットした値が無効である（0～2 以外の値の場合）

●ファイルの読み取り［function　3FH（63）：readh］

【マクロ定義】

```
READH   MACRO   BUFF, BYTE, HANDLE
        MOV     DX, OFFSET  BUFF  ; リード・バッファ
```

```
        MOV     CX, BYTE            ;リードするバイト数
        MOV     BX, HANDLE          ;オープンされているハンドル番号
        MOV     AH, 3 FH
        INT     21 H
        ENDM
```

【機能】

ファイル・ハンドル番号 HANDLE でオープンされているファイルから BYTE で示されるバイト数のデータをバッファ BUFF にリードします.

このときに，指定した BYTE 数が常にリードされるとは限らず，レジスタ AX に返される値が実際にリードされた値です．AX が 0 ならばファイルの終わりに達したことを表します.

リターン時にキャリーがセットされていると，リードは正常に行われなかったことになり，レジスタ AX には次のエラー・コードがセットされます.

AX＝5 ：BX で渡されたハンドラはリードが許可されていないモードである
　　6 ：BX で渡されたハンドラは現在オープンされていない

●ファイルのクローズ［function　3 EH（62）：closeh］

【マクロ定義】

```
CLOSEH  MACRO   HANDLE
        MOV     BX, HANDLE      ;ハンドル番号
        MOV     AH, 3 EH
        INT     21 H
        ENDM
```

【機能】

ファイル・ハンドル番号 HANDLE でオープンされているファイルをクローズします．これでこのファイル・ハンドル番号が開放されます.

クローズが正常に行われないとキャリーがセットされ，レジスタ AX にエラー・コードがセットされます.

AX＝6 ：BX で渡されたハンドラは現在オープンされていない

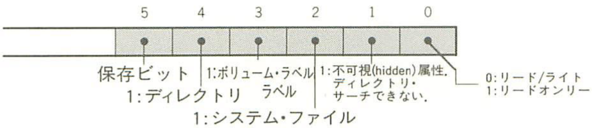
**●ファイルの生成［function　3 CH（60）：creath］**

**【マクロ定義】**

```
CREATEH  MACRO   FNAME, ATR
         MOV     DX, OFFSET  FNAME ；パス名の位置
         MOV     CX, ATR           ；ファイル属性
         MOV     AH, 3 CH
         INT     21 H
         ENDM
```

**【機能】**

FNAME で示されるパス名のファイルを新規作成します．または既存のファイルからデータを書き出すために，そのファイルの大きさを0にします．新規作成された場合には ATR で示されるファイルの属性がファイルに与えられます．ファイルの属性は次のビットで決まります．




リターン時にキャリーがセットされていなければ，レジスタ AX にファイル・ハンドル番号が返されます．

オープンが正常に行われないとキャリーがセットされ，レジスタ AX にエラー・コードがセットされます．

AX＝3 ：パス名のファイルがない
　　4 ：すでにオープンされているファイルが多すぎる
　　5 ：アクセスが否定された

**●ファイルへの書き込み［function　40 H（64）：writeh］**

**【マクロ定義】**

```
WRITEH  MACRO   BUFF, BYTE, HANDLE
        MOV     DX, OFFSET  BUFF ；バッファ
        MOV     CX, BYTE         ；ライトするバイト数
        MOV     BX, HANDLE       ；オープンされているハンドル番号
```

```
MOV     AH, 40 H
INT     21 H
ENDM
```

【機能】

ファイル・ハンドル番号 HANDLE でオープン(新規作成)したファイルにバッファ BUFF のデータを BYTE で示されるバイト数だけ書き込みます．実際に書き込んだバイト数はレジスタ AX に返されますが，その値が指定したバイト数と異なればエラーと見なされます．書き込みが正常でない場合は，キャリーがセットされてレジスタ AX にエラーコードがセットされます．

AX＝5 ：アクセスが否定された
　　6 ：BX で渡されたハンドラは現在オープンされていない

# 8.9／ファイル・ハンドラ・ヘッダの作成

前節 8.8 で説明したファイル・ハンドラを介して行うファイル入出力関係のマクロを**リスト 8.3** fileh.h ファイルとして作成することにします.

●リスト8.3　fileh.h（ファイル・ハンドラを介したファイル入出力ヘッダ）

```
; ------------------------------
; *     file handler header     *
; ------------------------------

OPENH    MACRO    FNAME,MODE
         MOV      DX,OFFSET FNAME
         MOV      AL,MODE
         MOV      AH,3DH
         INT      21H
         ENDM

READH    MACRO    BUFF,BYTE,HANDLE
         MOV      DX,OFFSET BUFF
         MOV      CX,BYTE
         MOV      BX,HANDLE
         MOV      AH,3FH
         INT      21H
         ENDM

CREATH   MACRO    FNAME,ATR
         MOV      DX,OFFSET FNAME
         MOV      CX,ATR
         MOV      AH,3CH
         INT      21H
         ENDM

WRITEH   MACRO    BUFF,BYTE,HANDLE
         MOV      DX,OFFSET BUFF
         MOV      CX,BYTE
         MOV      BX,HANDLE
         MOV      AH,40H
         INT      21H
         ENDM

CLOSEH   MACRO    HANDLE
         MOV      BX,HANDLE
         MOV      AH,3EH
         INT      21H
         ENDM
```

# 8.10 / fileh.hファイルの利用例

fileh .hファイルをユーザプログラムの先頭で,

```
INCLUDE  FILEH.H
```

と INCLUDE 擬似命令でヘッダ・ファイルとして取り込めば，ファイルの入出力処理に関するユーザの負担は軽くなり，高級言語の記述に近い表現が可能となります．ここでは fileh. h ファイルを用いた簡単なプログラムを例示することにします．

なおマクロコール部は，それがマクロであることがはっきりするように小文字で記述しました．

## TYPE コマンドの作成

**プログラム**

```
; ------------------------------------------
; *      type command by file handle      *
; ------------------------------------------

INCLUDE B:STDIO.H
INCLUDE B:FILEH.H

        dseg
MSG     DB      'filename ? $'
CONBUF  DB      32,33 DUP(?)      ; filename store
HANDLE  DW      ?                 ; handler number store
DSKBUF  DB      1024 DUP(?)       ; disk r/w buffer
        dend

        sseg              ; stack segment

        pseg
START:  MOV     AX,DATA
        MOV     DS,AX

        print   MSG               ; filename input             ①
        input   CONBUF
        MOV     BL,CONBUF[1]      ; set 0 string end           ②
        XOR     BH,BH
        MOV     BYTE PTR CONBUF[BX+2],0

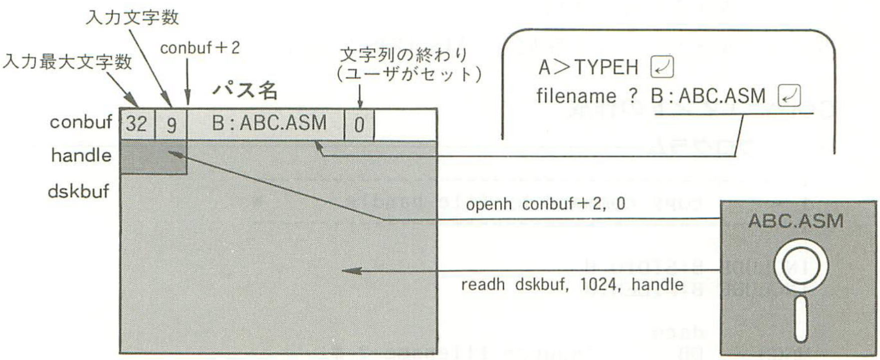
        openh   CONBUF+2,0        ; open file                  ③
        JC      FIN
```

```
        MOV     HANDLE,AX           ; filehandle store      ↓③

NEXT:   readh   DSKBUF,1024,HANDLE                          ④
        CMP     AX,0                ; file read end
        JE      FIN
        MOV     CX,AX               ; disp
        MOV     BX,OFFSET DSKBUF
DISP:   CMP     BYTE PTR [BX],1AH    ; text end mark         ⑤
        JE      FIN
        putchar [BX]
        INC     BX
        LOOP    DISP
        JMP     NEXT

FIN:    closeh  HANDLE              ; close file             ⑥
        retms
        pend
        END     START
```

図8.5　ファイル・ハンドラによるTYPEコマンドの概念




ファイル·ハンドラによるTYPEコマンドの概念を**図 8.5** に示します．基本的な考え方は以下の通りです．

・パス名は function 0 AH（input）により入力バッファ conbuf に格納する．入力した文字数（conbuf＋1に格納されている）を元にして，ユーザが文字列のの最後に 0 をセットする．

・conbuf＋2 がパス名の位置となるので，ファイルを読み込みモード（モー

ド0）でオープンします．ハンドル番号がレジスタ AX に返されるので，これを handle 番地に格納します．

・読み込みバッファ dskbuf のサイズは 1024 バイトとします．

それでは実際のプログラムの説明を行います．

①パス名を conbuf に入力する．

②パス名の最後に 0 （ヌル）をセットする．

③パス名で示されるファイルをオープンし，ファイル・ハンドル番号を handle 番地にセットする．ファイルがオープンできなければ終了します．

④ handle で示されるファイルを dskbuf にリードする．

⑤レジスタ AX には読み込んだバイト数が入っているので，その回数だけ dskbuf のデータをスクリーンに表示していく．テキストの終わりにエンド・マークとして 1AH が入っているが，これは表示しない．もし 1AH がスクリーンに表示されると画面クリアになる．

⑥ファイル・ハンドルを開放し，MS-DOS に戻る．

## COPY コマンドの作成

**プログラム**

```
; ----------------------------------------
; *      copy command by file handle      *
; ----------------------------------------

INCLUDE B:STDIO.H
INCLUDE B:FILEH.H

        dseg
MSGS    DB      'source filename ? $'
MSGD    DB      'destinate filename ? $'
CONBUFS DB      32,33 DUP(?)    ; source filename store
CONBUFD DB      32,33 DUP(?)    ; destinate filename store
HANDLES DW      ?               ; source handler number
HANDLED DW      ?               ; destinate handler number
DSKBUF  DB      1024 DUP(?)     ; disk r/w buffer
        dend

        sseg            ; stack segment

        pseg
START:  MOV     AX,DATA
        MOV     DS,AX
```

```
        print   MSGS            ; source filename input        ┬
        input   CONBUFS                                        │
        MOV     BL,CONBUFS[1]                                  │
        XOR     BH,BH                                          │ ①
        MOV     BYTE PTR CONBUFS[BX+2],0                       │
        openh   CONBUFS+2,0                                    │
        JC      FIN              ; file can't open             │
        MOV     HANDLES,AX                                     ┴

        print   MSGD            ; destinate filename input     ┬
        input   CONBUFD                                        │
        MOV     BL,CONBUFD[1]                                  │
        XOR     BH,BH                                          │ ②
        MOV     BYTE PTR CONBUFD[BX+2],0                       │
        creath  CONBUFD+2,0                                    │
        JC      FIN              ; file can't creat            │
        MOV     HANDLED,AX                                     ┴

NEXT:   readh   DSKBUF,1024,HANDLES                            ┬
        CMP     AX,0            ; file read end                │
        JE      FIN                                            │ ③
        writeh  DSKBUF,AX,HANDLED                              │
        JMP     NEXT                                           ┴

FIN:    closeh  HANDLES                                        ┬
        closeh  HANDLED                                        │ ④
        retms                                                  ┴
        pend
        END     START
```

基本的には先のTYPEコマンドと同じですが，読み込み用と書き込み用に2つのファイル・ハンドルを同時にオープンしている点が異なります．読み込み用のパス名をconbufs，ファイル・ハンドルをhandlesにセットし，書き込み用のパス名をconbufd，ファイル・ハンドルをhandledにセットします．

プログラムの説明は次の通りです．

①コピーのソース・ファイル名をconbufsに入力し，ファイルをオープンする．オープンされたファイル・ハンドル番号をhandlesにセットする．
②コピーのディスティネイト・ファイル名をconbufdに入力し，ファイルを作成する．オープンしたファイル・ハンドル番号をhandledにセットする．
③handlesで示されるファイルをリードし，handledで示されるファイルに書き込む．これをファイルの終わりまで繰り返す．
④ファイル・ハンドルを開放し，MS-DOSに戻る．

# 8.11 割り込み

MS-DOSでは20H～3FHまでの割り込みタイプが予約してあり，80H～FCHに割り込みルーチン・アドレス・テーブルを持っています．

MS-DOSが予約している割り込みを**表8.1**に示します．

●表8.1　MS-DOSの割り込みタイプ

| 割り込みタイプ | 機　能 |
|---|---|
| 20H | プログラムの終了 |
| 21H | ファンクション・リクエスト |
| 22H | 終了アドレス(プログラムでは利用できない) |
| 23H | <CTRL-C>の抜け出しアドレス(プログラムでは利用できない) |
| 24H | 致命的なエラーによる打ち切りアドレス(プログラムでは利用できない) |
| 25H | アブソリュート・ディスク・リード |
| 26H | アブソリュード・ディスク・ライト |
| 27H | プログラムを在駐させたまま終了 |
| 28H～3FH | 予備(ユーザは使用してはいけない) |

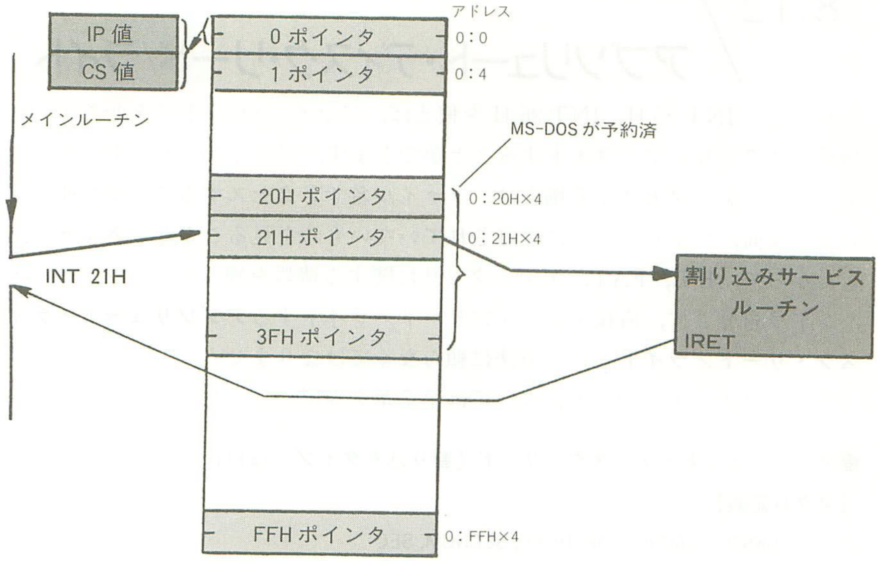
この割り込みタイプの中でユーザが利用できるのは，**割り込みタイプ20H，21H，25H，26H，27H**だけです．

タイプ21Hのファンクション・リクエストは既に説明したものです．次節以後にタイプ25H，26Hのアブソリュート・ディスク・リード／ライトの割り込みについて説明します．

**図8.6**に割り込みポインタ・テーブルのマップを示します．メモリの0：0から4バイト単位に256種のポインタが確保されています．各ポインタはIP値，CS値の順にエントリ・ポイント（ジャンプのためのアドレス値）が格納されています．

タイプ0～4ポインタは8086が既に予約しています．

図8.6　割り込みポインタ・テーブル




●割り込みポインタ・テーブル
（RAM 先頭0：0より確保されている）

# 8.12 アブソリュート・ディスク・リード/ライト

割り込みINT 25 H，INT 26 Hを使えば，ファイル・ハンドラを介さず，直接ディスクにリード／ライトすることができます．ファイル・ハンドラを介してファイルをアクセスする場合は，ファイル名でアクセスすることが可能で使い易い反面，ディレクトリに登録されていないものを見ることができません．したがって，IPL，FAT，ディレクトリに関する情報を知るには，ファイル・ハンドラを介さず，直接ディスクにリード／ライトする**アブソリュート・ディスク・リード／ライト**という方法に頼らなくてはなりません．

ディスクのフォーマットについては第7章を参照してください．

**●アブソリュート・ディスク・リード［割り込みタイプ　25 H］**

**【マクロ定義】**

```
DSKRD   MACRO   DR, BUFF, SECNUM, SEC
        MOV     AL, DR          ；ドライブ番号（0＝A，1＝B…）
        MOV     BX, BUFF        ；ディスク転送アドレス
        MOV     CX, SECNUM      ；読み込みセクタ数
        MOV     DX, SEC         ；読み込み開始相対セクタ
        INT     25 H
        POPF
        ENDM
```

**【機能】**

指定されたドライブ番号DRのディスクの相対セクタ番号SECのセクタからSECNUM個のセクタをBUFFに読み込みます．

セグメント・レジスタ以外のすべてのレジスタの内容は，この割り込みにより破壊されているので，保存の必要のあるレジスタはユーザの責任で退避させておかなければなりません（スタックにPUSHしておく）．

また，この割り込みでは，自動的にフラグがスタックに積まれて（PUSHF）リターンするので，ユーザはINT 25 H終了後にそのスタックを取り出す

（POPF）必要があります．

キャリー・フラグがセットされなければ，読み込みは正常に行われたことになり，キャリー・フラグがセットされると，リード・エラーを意味し，エラー・コードがレジスタ AL に与えられます．

**●アブソリュート・ディスク・ライト［割り込みタイプ　26 H］**

**【マクロ定義】**

```
DSKWR   MACRO   DR, BUFF, SECNUM, SEC
        MOV     AL, DR          ；ドライブ番号（0 = A，1 = B…）
        MOV     BX, BUFF        ；ディスク転送アドレス
        MOV     CX, SECNUM      ；読み込みセクタ数
        MOV     DX, SEC         ；読み込み開始相対セクタ
        INT     26 H
        POPF
        ENDM
```

**【機能】**

指定されたドライブ番号 DR のディスクの相対セクタ番号 SEC のセクタから SECNUM 個のセクタに BUFF のデータを書き込みます．

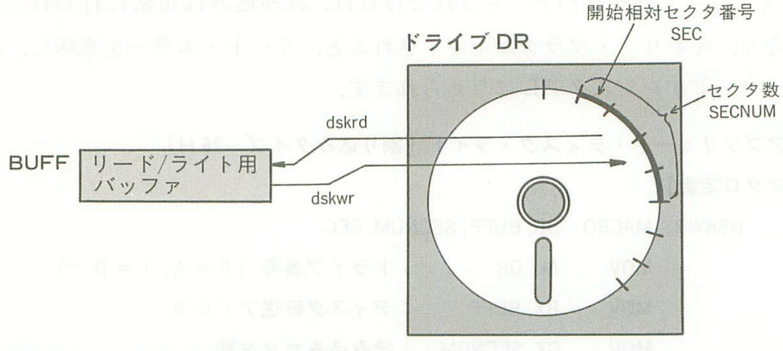
セグメント・レジスタ以外のすべてのレジスタの内容は，この割り込みにより破壊されているので，保存の必要のあるレジスタはユーザの責任で退避させておかなければなりません（スタックに PUSH しておく）．

また，この割り込みでは，自動的にフラグがスタックに積まれて（PUSHF）リターンするので，ユーザは INT 26 H 終了後にそのスタックを取り出す（POPF）必要があります．

キャリー・フラグがセットされなければ，書き込みが正常に行われたことになり，キャリー・フラグがセットされると，ライト・エラーを意味し，エラー・コードがレジスタ AL に与えられます．

アブソリュート・ディスク・リード／ライトの概念を**図 8.7** に示します．また，アブソリュート・ディスク・リード／ライトのマクロ定義ファイルを dskrw .h として**リスト 8.4** に示します．

図8.7 アブソリュート・ディスク・リード／ライト




●リスト8.4 dskrw.h（アブソリュート・ディスク・リード／ライト用ヘッダ）

```
; --------------------------------------
; *   absolute disk read/write header   *
; --------------------------------------

DSKRD   MACRO   DR,BUFF,SECNUM,SEC
        MOV     AL,DR               ; drive no.
        MOV     BX,OFFSET BUFF      ; read buffer
        MOV     CX,SECNUM           ; read sector number
        MOV     DX,SEC              ; start sector
        INT     25H
        POPF
        ENDM

DSKWR   MACRO   DR,BUFF,SECNUM,SEC
        MOV     AL,DR               ; drive no.
        MOV     BX,OFFSET BUFF      ; write buffer
        MOV     CX,SECNUM           ; write sector number
        MOV     DX,SEC              ; start sector
        INT     26H
        POPF
        ENDM
```

# 8.13／アブソリュート・ディスク・リード/ライトの利用例

**リスト 8.4** に示したdskrw.hによるアブソリュート・ディスク・リード／ライトを用いた簡単なプログラム例をを示します.

### ●ディスク・コピープログラム

**プログラム**

```
; -----------------------------
; *     disk copy program     *
; -----------------------------

INCLUDE B:STDIO.H
INCLUDE B:DSKRW.H                     Ⓐ

SECBYTE EQU     512   Ⓑ    ; 5"2DD sector size
SECNUM  EQU     4          ; R/W sector number
MAXSEC  EQU     4FFH+1     ; max sector
BUFSIZE EQU     SECBYTE*SECNUM

        dseg               ; data segment
MSG     DB      'drive A --> drive B',CR,LF
        DB      '   hit anykey$'
BUFF    DB      BUFSIZE DUP(?)  ─────────── ①
        dend

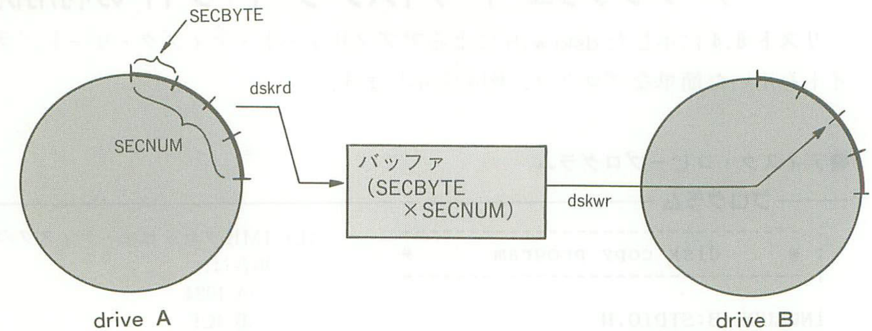
        sseg               ; stack segment

        pseg
START:  MOV     AX,DATA
        MOV     DS,AX
        print   MSG                          ┐
HIT:    inkey                                │ ②
        CMP     AL,0                         │
        JE      HIT                          ┘
        MOV     DX,0  ───────────────────── ③
        MOV     CX,MAXSEC/SECNUM             ┐
NEXT:   PUSH    CX                           │
        dskrd   0,BUFF,SECNUM,DX   ; drive A read
        dskwr   1,BUFF,SECNUM,DX   ; drive B write   ④
        ADD     DX,SECNUM                    │
        POP     CX                           │
        LOOP    NEXT                         ┘
        retms
        pend
        END     START
```

注）1MBフロッピィ・ディスクの場合は,
Ⓐ 1024
Ⓑ 4CF
に変更する.

ディスクの内容をセクタ単位でそっくりコピーするプログラムです．図 8.8 にその概念図を説明します．

図8.8 ディスク·コピー




コピーは drive A → drive B に向かって行われ，1回のディスクの読み込み／書き込みのセクタ数を SECNUM，1セクタ当たりのバイト数を SECBYTE（5”2DD では 512 バイト，8”2D または 5”2HD では 1024 バイト）に与えます．したがって読み込み／書き込み用のバッファのサイズは SECNUM×SECBYTE となります．

読み込み／書き込みの開始セクタは 0 セクタから始め，1回の読み込み／書き込みのたびに SECNUM の値だけ増加します．

ディスクの最大セクタ番号（5”2DD では 4FFH，8”2D または 5”2HD では 4CFH）を MAXSEC に与えます．したがって，読み込み／書き込みの繰り返しは MAXSEC／SECNUM となります．

プログラムの説明を次に示します．

①読み込み／書き込み用のバッファ
②メッセージの表示とキー入力．いずれかのキーが押されるとコピーを開始する
③開始セクタを 0 に初期設定する
④ドライブA→ドライブBへのディスク・コピーを行う．繰り返し数 CX はマクロ dskrd，dskwr で破壊されるので PUSH／POP しなければなりません．

# 第9章
# エスケープ・シーケンス

ESC

## 本章の目的

前章で示したファンクションコール（システムコール）の中には，スクリーンをカラー・モードにしたり，スクリーンの任意の位置にカーソルを移動するといった機能は含まれてはいません．それは，CRT スクリーンの画面表示のようなカーソル・アドレッシングに関する事柄は，ハードウェアに依存しているためです．

このようなハードウェアに依存する部分を OS の BIOS 部(IO. SYS)にまとめ，ある規格化されたコード文字列をスクリーンに送ることにより，スクリーンを制御することをエスケープ・シーケンス呼びます．

ここでは，エスケープ・シーケンスを用いて，BASIC でなじみのある color，locate，tab などの機能を作成することにします．

さらにプリンタに関するエスケープ・シーケンスを用いて，プリンタの印字モードを変更するプログラムも説明します．

# 9.1／エスケープ・シーケンスの種類

エスケープ・シーケンスは，ESC コード（1 BH）で始まる何文字かの文字列からなります．

エスケープ・シーケンスの文字列は使用するコンピュータのハードウェアにより多少異なる場合もあるので，詳しくは MS-DOS のマニュアルを参照してください．ここでは，NEC PC-9800 シリーズ用の MS-DOS および CP／M-86 のエスケープ・シーケンスを**表 9.1** に示します．

●表9.1　エスケープ・シーケンス・コード

| | エスケープ・シーケンス | 機　能 |
|---|---|---|
| カーソル移動 | ESC[pl;pcH | カーソルを(pc, pl)位置に移動．pcが x 座標，plが y 座標 |
| | ESC[pn A | カーソルを pn 行アップ |
| | ESC[pn B | カーソルを pn 行ダウン |
| | ESC[pn C | カーソルを右に pn 文字移動 |
| | ESC[pn D | カーソルを左に pn 文字移動 |
| | ESC D | カーソルを 1 行下に移動 |
| | ESC E | ニューライン（'\n' または cr-lf） |
| | ESC M | カーソルを 1 行上に移動 |
| | ESC[s | カーソル位置(行：y, カラム：x)とその表示文字の属性をセーブ |
| | ESC[u | ESC[s でセーブした内容を戻す |
| | ESC[6n | カーソル位置を知らせる |
| 文字・行の削除／挿入 | ESC[0J | カーソル位置から最終行の右端までをクリア |
| | ESC[1J | 先頭行の左端からカーソル位置までをクリア |
| | ESC[2J | スクリーンのクリア．ESC*でも良い |
| | ESC[0K | カーソル位置から行の右端までをクリア |
| | ESC[1K | 行の左端からカーソル位置までをクリア |
| | ESC[2K | カーソル行の文字をすべてクリア |
| | ESC[pn M | pn 行削除 |
| | ESC[pn L | pn 行分の空白行を挿入 |
| 画面モード | ESC) 0 | 漢字モード（システム既定値） |
| | ESC) 3 | グラフ文字表示モード |
| | ESC[>5l | カーソル表示モード（システム既定値） |
| | ESC[>5h | カーソル非表示モード |
| | ESC[>1h | 最下行をプログラムで使用可にする |

| | | | | |
|---|---|---|---|---|
| の設定 | ESC[>1l | 最下行をシステムで使用する（システム既定値） | | |
| | ESC[>3h | 1画面20行モード | | |
| | ESC[>3l | 1画面25行モード（システム既定値） | | |
| 文字の属性の設定 | ESC[ps;…;ps m] | 表示文字の属性を指示 | ps | 属性 |
| | | 【例】　ESC,'[34m'……青 | 0 | 既定値 |
| | | | 1 | ハイライト |
| | | ESC,'[m'……既定値に戻す | 5 | ブリンク |
| | | | 7 | リバース |
| | | 注）30～37の各色は各々40～47で反転文字の表示が行える． | 8(16) | シークレット |
| | | | 30 | 黒 |
| | | | 34(18) | 青 |
| | | | 31(17) | 赤 |
| | | | 35(19) | 紫 |
| | | | 32(20) | 緑 |
| | | | 36(22) | 水色 |
| | | | 33(21) | 黄色 |
| | | | 37(23) | 白 |

スクリーン制御コードとしては，エスケープ・シーケンス以外にも1文字のASCII制御コードも用意されています．その例を次の**表9.2**に示します．このASCII制御コードは各ハードウェアにより異なるので，各機種のマニュアルを参考にしてください．

●表9.2　ASCII制御コード

| コード | 機　能 |
|---|---|
| 07 | ブザーを1秒鳴らす |
| 08 | カーソルを1文字左に移動 |
| 09 | カーソルを次のタブ位置に移動 |
| 0A | ラインフィード（改行） |
| 0B | カーソルを上へ1行移動 |
| 0C | カーソルを1文字右に移動 |
| 0D | リターン（復帰） |
| 1A | スクリーンのクリア |
| 1B | エスケープコード |
| 1E | カーソルホーム |

# 9.2 / カラー指定(color)

●黒（0）〜白（7）までの8色のカラー指定を行う color 機能を作成する

## プログラム

```
; -------------------------
; *      color n          *
; -------------------------
INCLUDE B:STDIO.H

COLOR   MACRO   COL
        LOCAL   FIN
        MOV     CL,COL
        CMP     CL,7              ; if COL>7 then fin
        JA      FIN
        MOV     DX,OFFSET BLACK   ; color data start offset
        MOV     AX,0006H          ; data size
        MUL     CL
        ADD     DX,AX             ; DX <-- BLACK+6*COL
        MOV     AH,9              ; function 9
        INT     21H
FIN:
        ENDM

        dseg
BLACK   DB      ESC,'[30m$'       ; color 0 (black)
        DB      ESC,'[18m$'       ; color 1 (blue)
        DB      ESC,'[17m$'       ; color 2 (red)
        DB      ESC,'[19m$'       ; color 3 (violet)
        DB      ESC,'[20m$'       ; color 4 (green)
        DB      ESC,'[22m$'       ; color 5 (light blue)
        DB      ESC,'[21m$'       ; color 6 (yellow)
        DB      ESC,'[m$'         ; color 7 (white:normal)

MSG     DB      'Hello MS-DOS world!',CR,LF,'$'
        dend

        sseg

        pseg
START:  MOV     AX,DATA
        MOV     DS,AX
        MOV     CX,7
NEXT:   color   CL
        print   MSG
        LOOP    NEXT
        color   7
        retms
        pend
        END     START
```

注記:
- CL が 7 を超えた場合は何もしない（CMP CL,7 ～ JA FIN）
- COL で示されるデータの先頭オフセットを DX に入れる（MOV DX,OFFSET BLACK ～ ADD DX,AX）
- CL の色で MSG を表示する（color CL ～ LOOP NEXT）
- 白にもどす（color 7）

カラー指定は，次のエスケープ・シーケンスをスクリーンに送ることにより行うことができます．

```
ESC, '[psm'
```

psが'30'なら黒，'18'なら青…となります．ここでは，

```
color  n
```

とマクロコールすることで，カラーコードに対応する色が得られるようにしてあります．

黒（0）～白（7）までのカラー・コードのデータを，BLACK以後に格納しておきます．データの終わりの'$'はファンクション9のprintを実行するときの文字列の終わりの印です．

マクロcolorの引数COLにカラー・コード（0～7）が与えられるので，これを元にしてカラー・コードのデータが格納されている領域の先頭アドレスを求め，レジスタDXにセットします．データは6バイトおきに格納されているので，

```
DX  ←  BLACK + 6 × COL
```

となります．以上を行った後，ファンクション9でレジスタDXに示される文字列をスクリーンに表示すればよいということになります．

なお，引数COLに7を超える値が入った場合は，何もしないでプログラムを抜け出すことにしてあります．

このプログラムはマクロcolorを利用して，'Hello MS-DOS world!'というメッセージを7色で表示するものです．

次のマクロコール

```
color  CL
```

で，レジスタCLが示すカラー・コードの色を指定します．

# 9.3 ／ 座標指定(locate)

●任意の座標にカーソルを移動する locate 機能を作成する

## プログラム

```
; -------------------------
; *    locate (x.y)       *
; -------------------------
INCLUDE B:STDIO.H

BCD      MACRO    NUM,ADR
         MOV      AL,NUM
         XOR      AH,AH
         MOV      CL,100            ; 3ケタメ
         DIV      CL
         ADD      AL,'0'
         MOV      BYTE PTR ADR,AL
         MOV      AL,AH             ; 2ケタメ
         XOR      AH,AH
         MOV      CL,10
         DIV      CL
         ADD      AL,'0'
         MOV      BYTE PTR (ADR+1),AL
         ADD      AH,'0'            ; 1ケタメ
         MOV      BYTE PTR (ADR+2),AH
         ENDM

LOCATE   MACRO    X,Y
         bcd      Y,LOCXY+2
         bcd      X.LOCXY+6
         print    LOCXY
         ENDM

         dseg
LOCXY    DB       1BH,'[000;000H$'  ; locate(y,x) table
MSG      DB       'Hello$'
         dend

         sseg               ; stack segment

         pseg
START:   MOV      AX,DATA
         MOV      DS,AX
         locate   30,10
         print    MSG
         retms
         pend
         END      START
```

（注記）
- MOV AL,NUM ～ MOV BYTE PTR ADR,AL：3桁目のアスキー文字をLOCXY＋2に格納する
- MOV AL,AH ～ MOV BYTE PTR (ADR+1),AL：2桁目のアスキー文字をLOCXY＋3に格納する
- ADD AH,'0' ～ MOV BYTE PTR (ADR+2),AH：1桁目のアスキー文字をLOCXY＋4に格納する
- bcd Y,LOCXY+2 ← Yの内容を3桁のアスキー文字列(10進)にして LOCXY＋2以後に格納する

任意の座標（pc，pl）にカーソルを移動するには，

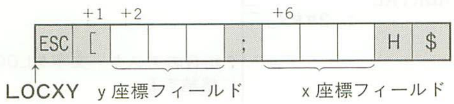
```
ESC，'［pl；pcH'
```

をスクリーンに送ることによって行われます．pc が x 座標，pl が y 座標に相当し，エスケープ・コードは y，x の順序で行います．たとえば，(20，10）の位置にカーソルを移動させるには，

```
ESC，'［10；20H'
```

とします．ここで注意しておかなければならないことは，20，10 をアスキー文字列で与えなければならないということです．

locate のためのエスケープ・シーケンスのデータを LOCXY 以後に次のように確保します．




まず，次のマクロコール

```
locate   X，Y
```

で与えられた引数 X，Y を 10 進数の文字列に変換し，そのデータを上図の LOCXY テーブルの x 座標フィールド，y 座標フィールドに 3 桁の 10 進アスキー文字列として格納するマクロが，bcd です．

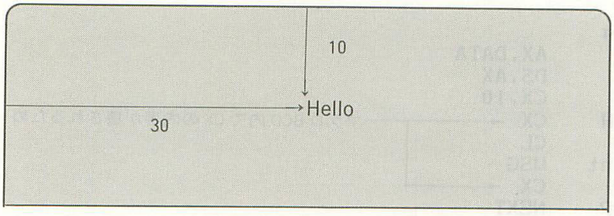
マクロ bcd は与えられた数 NUM を 3 桁の 10 進アスキー文字列にして ADR 番地以後に格納します．マクロ bcd の機能は次の通りです．

NUM を 100 で割った商＋'0'を ADR 番地に格納し，次に余りを 10 で割った商＋'0'を ADR＋1 番地に格納し，最後に余り＋'0'を ADR＋2 番地に格納します．このプログラムの中の

```
locate   30，10
print    MSG
```

という命令が実行されると，次に示すような CRT スクリーン上の位置に文字

列が表示されます．




# 9.4 ／ タブ指定(tab)

●CRTの左端からの位置指定を行うtab機能を作成する

## プログラム

```
; --------------------------
; *       tab(x)           *
; --------------------------
INCLUDE B:STDIO.H

BCD         MACRO    NUM,ADR
            MOV      AL,NUM
            XOR      AH,AH
            MOV      CL,100              ; 3ケタメ
            DIV      CL
            ADD      AL,'0'
            MOV      BYTE PTR ADR,AL
            MOV      AL,AH               ; 2ケタメ
            XOR      AH,AH
            MOV      CL,10
            DIV      CL
            ADD      AL,'0'
            MOV      BYTE PTR (ADR+1),AL
            ADD      AH,'0'              ; 1ケタメ
            MOV      BYTE PTR (ADR+2),AH
            ENDM

TAB         MACRO    X
            putchar  0DH                 ; cr
            bcd      X,RMOVE+2           ; rmove table set
            print    RMOVE               ; cursole right move
            ENDM

            dseg
RMOVE       DB       ESC,'[000C$'        ; tab(x) table
MSG         DB       'Hello',CR,LF,'$'
            dend
```

カーソルをスクリーンの左端に移動（cr）

Xの内容をアスキー文字列にしてRMOVE＋2以後に格納する（rmove table set）

RMOVEテーブルにセットされたデータだけ，カーソルを右に移動（cursole right move）

```
        sseg                    ; stack segment

        pseg
START:  MOV     AX,DATA
        MOV     DS,AX
        MOV     CX,10
NEXT:   PUSH    CX  ←────┐マクロBCD内でCXの内容が壊されるため
        tab     CL       │
        print   MSG      │
        POP     CX  ←────┘
        LOOP    NEXT
        retms
        pend
        END     START
```

---

マクロコール

tab　x

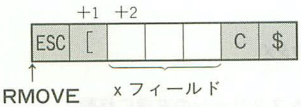
により，左端からxの位置にカーソルを移動するには，

putchar　0DH

でカーソルを現在ある行の左端に移動した後，

ESC,‘[xC’

をスクリーンに送ることによりx文字分だけカーソルを右に移動させるという操作を行います．x文字分だけカーソルを右に移動させるためのエスケープ・シーケンスのデータをRMOVE以後に次のように確保します．




このフィールドへデータをセットするには，locateの場合と同様にマクロbcdを用いて行います．

このプログラムの実行結果は次のページの図のようになります．

▼tabの実行結果

```
A>B:TAB
          Hello
         Hello
        Hello
       Hello
      Hello
     Hello
    Hello
   Hello
  Hello
 Hello
```

# 9.5／screen.hファイルの作成

エスケープ・シーケンスによるスクリーン制御のマクロ定義ファイルをscreen.hとして**リスト 9.1**に示します．

先に説明したcolor，locate，tab機能以外に次のマクロを追加しました．

● beep

ビープ音を出します．

● cls

スクリーンをクリアします．

● home

カーソルをホーム・ポジションに移動します．

● width 20

スクリーンに表示できる行を20行にします．

● width 25

スクリーンに表示できる行を25行にします（システム既定値）．

● escdat

エスケープ・シーケンスのデータを各種定義します．

●リスト9.1 screen.h（スクリーン制御ヘッダ）

```
;  -------------------------------------
;  *      screen interface header      *
;  -------------------------------------

BCD     MACRO   NUM,ADR
        MOV     AL,NUM
        XOR     AH,AH
        MOV     CL,100              ; 3ｹﾀﾒ
        DIV     CL
        ADD     AL,'0'
        MOV     BYTE PTR ADR,AL
        MOV     AL,AH               ; 2ｹﾀﾒ
        XOR     AH,AH
        MOV     CL,10
        DIV     CL
        ADD     AL,'0'
        MOV     BYTE PTR (ADR+1),AL
        ADD     AH,'0'              ; 1ｹﾀﾒ
        MOV     BYTE PTR (ADR+2),AH
        ENDM

LOCATE  MACRO   X,Y                 ;       locate (x,y)
        bcd     Y,LOCXY+2
        bcd     X,LOCXY+6
        print   LOCXY
        ENDM

TAB     MACRO   X                   ;       tab(x)
        putchar 0DH                 ; cr
        bcd     X,RMOVE+2           ; rmove table set
        print   RMOVE               ; cursole right move
        ENDM

COLOR   MACRO   COL                 ;       color n
        LOCAL   FIN
        MOV     CL,COL              ; if COL>7 then fin
        JA      FIN
        MOV     DX,OFFSET BLACK ; color data start offset
        MOV     AX,0006H            ; data size
        MUL     CL
        ADD     DX,AX               ; DX <-- BLACK+6*COL
        MOV     AH,9                ; function 9
        INT     21H
FIN:
        ENDM

BEEP    MACRO
        putchar 7
        ENDM

CLS     MACRO
        putchar 1AH
        ENDM
```

```
HOME     MACRO
         putchar 1EH
         ENDM

WIDTH20  MACRO                          ; 20 line
         print   W20
         ENDM

WIDTH25  MACRO                          ; 25 line
         print   W25
         ENDM

ESCDAT   MACRO
LOCXY    DB      ESC,'[000;000H$'; locate(y,x) table
RMOVE    DB      ESC,'[000C$'     ; tab(x) table

BLACK    DB      ESC,'[30m$'      ; color 0 (black)
         DB      ESC,'[18m$'      ; color 1 (blue)
         DB      ESC,'[17m$'      ; color 2 (red)
         DB      ESC,'[19m$'      ; color 3 (violet)
         DB      ESC,'[20m$'      ; color 4 (green)
         DB      ESC,'[22m$'      ; color 5 (light blue)
         DB      ESC,'[21m$'      ; color 6 (yellow)
         DB      ESC,'[m$'        ; color 7 (white:normal)

W20      DB      ESC,'[>3h$'      ; 20 line table
W25      DB      ESC,'[>3l$'      ; 25 line table
         ENDM
```

# 9.6 screen.hを用いたスクリーン制御例

**リスト 9.1** で示した screen.h をユーザプログラムで

```
INCLUDE   SCREEN.H
```

として取り込むことで，スクリーン制御に関するユーザの負担を軽くすることができます．

データ・セグメントで

```
escdat
```

とエスケープ・シーケンスのデータのマクロをコールしておくことに注意してください．以下に具体例を示します．

**プログラム**

```
; ----------------------------------------
; *       screen.h liblary example       *
; ----------------------------------------
INCLUDE B:STDIO.H
INCLUDE B:SCREEN.H

        dseg
        escdat              ; esc data macro
MSG     DB      'Hello MS-DOS world!',CR,LF,'$'
        dend

        sseg                ; stack segment

        pseg
START:  MOV     AX,DATA
        MOV     DS,AX
        cls   <-------------------------------- ①
        width20<------------------------------- ②
        color   5 <---------------------------- ③
        MOV     CX,7 ---------------┐
NEXT:   PUSH    CX                  │
        tab     CL                  │ ④
        print   MSG                 │
        POP     CX                  │
        LOOP    NEXT ---------------┘
        beep  <-------------------------------- ⑤
        color   7
        retms
        pend
        END     START
```

**【プログラム・リストの説明】**

①画面クリア

②スクリーンを 20 行モードにセットする

③表示色をライト・ブルーにする

④ tab 指定しながら MSG の内容を 7 回表示

⑤ビープ音を出す

# 9.7 / プリンタの印字モードの設定

スクリーンに関するエスケープ・シーケンスがあるように，プリンタに関するエスケープ・シーケンスもあります．ここでは，その中からプリンタの印字モードを設定するものをプログラム例として示します．

**プログラム**

```
; ------------------------------------------
; *      printer mode change              *
; ------------------------------------------

INCLUDE B:STDIO.H

        dseg
HS      DB      ESC,'N$'            ; HS pica
HD      DB      ESC,'H$'            ; HD pica
COND    DB      ESC,'Q$'            ; condence
ELITE   DB      ESC,'E$'            ; elite
PROP    DB      ESC,'P$'            ; proportional
LARGE   DB      0EH,'$'             ; ｶｸﾀﾞｲ
SMALL   DB      0FH,'$'             ; ｶｸﾀﾞｲ ｶｲｼﾞｮ
MSG     DB      'Hello MS-DOS world !',CR,LF,'$'
        dend

        sseg                ; stack segment

        pseg
START:  MOV     AX,DATA
        MOV     DS,AX
        lprint  HS          ; HS pica
        lprint  MSG
        lprint  HD          ; HD pica
        lprint  MSG
        lprint  COND        ; condence
        lprint  MSG
        lprint  ELITE       ; elite
        lprint  MSG
        lprint  PROP        ; proportional
```

```
lprint  MSG
lprint  LARGE    ; ｶｸﾀﾞｲ
lprint  MSG
lprint  SMALL    ; ｶｸﾀﾞｲ ｶｲｼﾞｮ
retms
pend
END     START
```

制御コードはプリンタの機種に依存するものですから，詳しくは各プリンタのマニュアルを参考にしてください．ここで例として用いた機種は，NECのPC-PR 201です．

たとえば，プリンタに対して

ESC，'N'

というコードを送ると，プリンタの印字モードはHS **パイカ**となります．

このプログラムの実行結果を次に示します．

▼プリンタモード切り換えプログラムの実行結果

```
Hello MS-DOS world !
Hello MS-DOS world !
Hello MS-DOS world !
Hello MS-DOS world !
Hello MS-DOS world !
Hello MS-DOS world !
```

# 第10章

# ユーティリティ・プログラム集

Program

**本章の目的**

前章までに説明した内容を発展させて，ちょっとしたユーティリティ・プログラムをいくつか紹介することにします．

ここで紹介するプログラムは，既に説明したSTDIO .H，FILE .H，DSKRW .H，SDREEN .Hや新たにここで作成するMYLIB .Hなどの各種マクロ定義ファイルを取り込むことで，プログラマの負担を軽減させています．

# 10.1／mylib.hヘッダ・フィイルの作成

これまでに説明したファンクションコールやエスケープ・シーケンス関係のマクロ定義ファイル以外にも，ユーティリティ・プログラムを作る際によく使われる処理をマクロ定義しておくと便利です．ここではそれらの機能をまとめたマクロ定義ファイルを mylib.h として**リスト 10.1** に示します．

●リスト10.1　mylib.h

```
; --------------------------
; *   マクロ・ライブラリ    *
; --------------------------

ASC_HEX MACRO   BUF
        LOCAL   BR1,BR2
        MOV     AX,WORD PTR BUF ┐①
        SUB     AL,'0'  ┐ ; ----------------------------------------
        CMP     AL,0AH  │ ; *   2桁のASCIIを1バイトの16進に   *
        JC      BR1     │ ; *     buf  : ASCII上位バイト          *
        SUB     AL,7    ②; *     buf+1: ASCII下位バイト          *
BR1:    MOV     CL,4    │ ; *     AL   : 16進                     *
        SHL     AL,CL   ┘ ; ----------------------------------------
        SUB     AH,'0'  ┐
        CMP     AH,0AH  │
        JC      BR2     ③
        SUB     AH,7    ┘
BR2:    OR      AL,AH   ┘④
        ENDM

HEX_ASC MACRO   HEX,BUF ; ----------------------------------------
        LOCAL   BR1,BR2 ; * 1バイトの16進を2桁のASCIIに  *
        MOV     AL,HEX  ; *   hex : 変換する16進                 *
        PUSH    AX      ; *   buf : 変換した上位バイト           *
        AND     AL,0F0H┐; * buf+1: 変換した下位バイト           *
        MOV     CL,4   │; ----------------------------------------
        SHR     AL,CL  ┘①
        ADD     AL,'0'
        CMP     AL,'9'+1           ┐
        JC      BR1                │
        ADD     AL,7               ②
BR1:    MOV     BYTE PTR BUF,AL    ┘
        POP     AX                   ┐
        AND     AL,0FH               │
        ADD     AL,'0'               │
        CMP     AL,'9'+1             ③
        JC      BR2                  │
        ADD     AL,7                 │
BR2:    MOV     BYTE PTR BUF+1,AL    ┘
        ENDM
```

```
HEX_DSP MACRO    HEX,BUF          ; -----------------------------------------
        hex_asc  HEX,BUF          ; *    １６進を２桁で表示                   *
        MOV      AH,2             ; *  hex : 表示する１６進                   *
        MOV      DL,BYTE PTR BUF  ; *  buf : 変換したＡＳＣＩＩの格納域       *
        INT      21H              ; -----------------------------------------
        MOV      DL,BYTE PTR BUF+1
        INT      21H
        ENDM

HEX_BCD MACRO    HEX,BUF          ; ---------------------------------------
        MOV      AL,HEX           ; *    １６進を２桁のＢＣＤに             *
        XOR      AH,AH            ; *  hex : １６進                         *
        MOV      CL,10            ; *  buf : 変換したＢＣＤの格納域         *
        DIV      CL               ; ---------------------------------------
        ADD      AL,'0'
        MOV      BYTE PTR BUF,AL
        ADD      AH,'0'
        MOV      BYTE PTR BUF+1,AH
        ENDM

DUMP    MACRO    BUF,PARA,WORK               ; -------------------------
        LOCAL    DUMPST,LINE1,ASC,NO,NUM     ; *    １６進ダンプ       *
        MOV      CX,PARA                     ; * buf  : 先頭アドレス   *
        MOV      BX,OFFSET BUF               ; * para : 表示バイト     *
DUMPST: PUSH     CX                          ; -------------------------
        MOV      CX,16         ; hex dump
LINE1:  MOV      AL,BYTE PTR [BX]  ┐
        PUSH     CX                │①
        hex_dsp  AL,WORK           ┘
        POP      CX
        putchar  SPACE
        INC      BX
        LOOP     LINE1
        SUB      BX,16    ]②             ; ascii dump
        MOV      CX,16
ASC:    MOV      DL,BYTE PTR [BX]  ┐
        CMP      DL,20H            │
        JB       NO                │③
        CMP      DL,7FH            │
        JB       NUM               │
NO:     MOV      DL,'.'            ┘
NUM:    putchar  DL                ]④
        INC      BX
        LOOP     ASC
        putchar  CR
        putchar  LF
        POP      CX
        LOOP     DUMPST
        ENDM

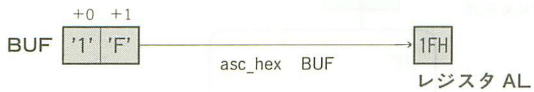
MOVDAT  MACRO    OFF1,OFF2,BYTE    ; ------------------------------
        MOV      CX,BYTE           ; *   データのブロック転送     *
        MOV      SI,OFFSET OFF1    ; * off1 : souce               *
        MOV      DI,OFFSET OFF2    ; * off2 : destinate           *
        PUSH     DS                ; * byte : 転送バイト          *
        POP      ES                ; ------------------------------
```

```
        CLD
        REP         MOVSB
        ENDM
```

---

このmylib .hは次のようなマクロ定義から構成されています．

**● asc_hex：2桁のアスキー・コードを1バイトの16進数に変換する**




BUFにアスキー・コードの上位バイト，BUF＋1に下位バイトを格納し，

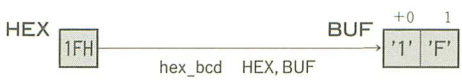
```
asc_hex  BUF
```

とマクロコールすると，レジスタALに1バイトの16進数が得られます．

【プログラムの説明】

①2桁のアスキー・コードをレジスタAXにロードする
②アスキー・コードの上位バイトをxxxx 0000の形の16進数に変換する
③アスキー・コードの下位バイトを0000 xxxxの形の16進数に変換する
④上位，下位を合成し，1バイトの16進数にする

**● hex_asc：1バイトの16進数を2桁のアスキー・コードに変換する**




HEX（レジスタやメモリなど）の16進数を2桁のアスキー・コードに変換し，BUFに上位，下位の順に格納します．マクロは，

```
hex_asc  HEX, BUF
```

とコールします．

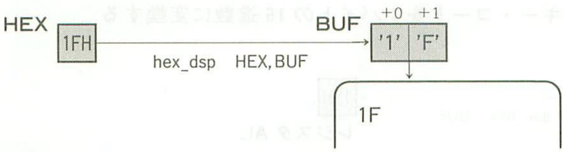
【プログラムの説明】

①上位4ビットを取り出し，0000 xxxxの形の16進数に変換する

②‘0’を加えて‘9’を超えるものは‘A’～‘F’となるように補正する．結果として得たアスキー・コードをBUFに格納する

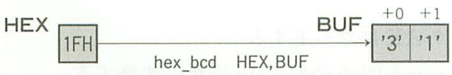
③下位4ビットについても同様の処理を行い，変換したアスキー・コードをBUF＋1に格納する

### ● hex_dsp：1バイトの16進数を2桁で表示する




先のhex_ascによりBUFに2桁のアスキー文字を格納し，このバッファの内容をputcharによりスクリーンへ表示します．

### ● hex_bcd：1バイトの16進数を2桁のBCDに変換する




先のhex_ascは16進表現のアスキー文字に変換したのに対して，このhex_bcdは10進表現のアスキー文字に変換します．つまり，1FHは‘1F’ではなく，‘31’に変換します．

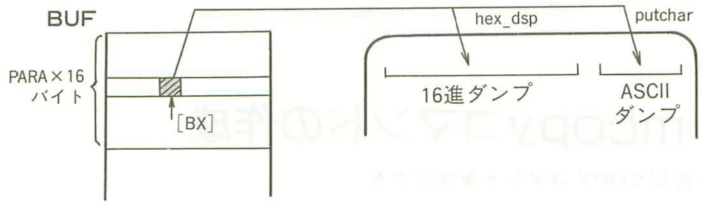
マクロコールは，

```
hex_bcd HEX, BUF
```

とします．ただし，このマクロは2桁のアスキー文字を生成するもので，100(64H) 以上の16進数に対しては無効です．

### ● dump：16進ダンプ

BUFが指す領域の先頭アドレスからPARAで示されるパラグラフ（para×16バイト）だけ，メモリの内容を16進とアスキー・コードでダンプ表示します．




マクロコールは，

```
dump    BUF,PARA
```

とします．

【プログラムの説明】

①[BX]の内容を16進で表示する

②16バイト分，アドレスを前に戻す

③アスキー・コード20H～7FH以外はピリオド（.）に置換する

④[BX]の内容をアスキー表示する

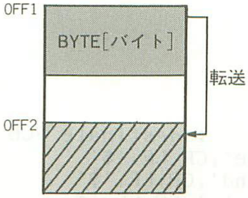
● movdat：データのブロック転送

データ・セグメント内のブロック転送を行います．オフセットのOFF1からBYTE分のデータをオフセットOFF2以後へ転送します．マクロコールは

```
movdat    OFF1,OFF2,BYTE
```

とします．

なお，この2つのデータの領域は重複してはなりません．



# 10.2 / mcopyコマンドの作成

●エラー処理を含むCOPYコマンドを作成する

## プログラム

```
; -----------------------------------------
; *    コピー・コマンド（エラー処理含む）      *
; *        A>MCOPY source destinate          *
; -----------------------------------------

INCLUDE B:STDIO.H
INCLUDE B:FILE.H
INCLUDE B:MYLIB.H

CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,SS:CODE
        ORG     5CH
FCB1    DB      32 DUP(?)        ; source FCB
        ORG     100H
START:  movdat  FCB1+16,FCB2,16 ; destinate FCB --> FCB2
        open    FCB1
        CMP     AL,0FFH          ; ファイルが見つからない
        JE      NOPEN
        create  FCB2
        CMP     AL,0FFH          ; ファイルが生成できない
        JE      NCREAT
        MOV     FCB1+32,0
        MOV     FCB2+32,0
COPY:   read    FCB1
        CMP     AL,1
        JE      FIN
        write   FCB2
        JMP     COPY
FIN:    close   FCB2
        CMP     AL,0FFH          ; ファイルがクローズできない
        JE      NCLOSE
        print   MSG0
        retms
NOPEN:  print   MSG1             ; エラー処理
        retms
NCREAT: print   MSG2
        retms
NCLOSE: print   MSG3
        retms

FCB2    DB      34 DUP(?)        ; destinate FCB
MSG0    DB      'copy complete',CR,LF,'$'
MSG1    DB      'file not found',CR,LF,'$'
MSG2    DB      'file not create',CR,LF,'$'
MSG3    DB      'file not close',CR,LF,'$'
```

```
CODE        ENDS
            END         START
```

注）EXE2BINによりCOMファイルに変換すること

第8章で示したプログラムはできるだけコンパクトなプログラムとするためにエラー処理までは考慮してありませんでした．しかし，一般にはファイル入出力関係でエラーが生じたときにそのままプログラムを進行させると，データなどに重大なダメージを与えてしまうことがあります．そこで，ファイルのオープン／作成(create)／クローズにおいてエラーが生じた場合に，それぞれのエラー・メッセージを出力してプログラムの実行を中止するようにしました．

本格的なコマンドにするには，これにエラー・リカバリ（修復）機能を付けることで完全なものになります．

また，第8章8.10のCOPYコマンドでは，ファイル名をコマンド・ラインから与える方式ではなかったので，ここではファイル名を次のようにコマンド・ラインから与えるものとします．

```
A>MCOPY  filename 1  filename 2 ↵
```

これによって図**10.1**のようにPSP内にパラメータが引き渡されます．

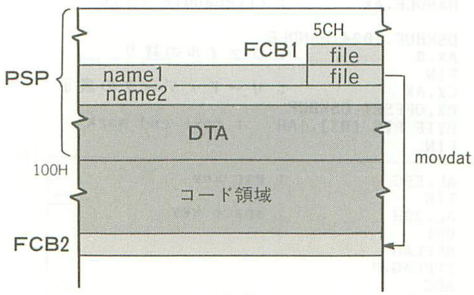



図10.1　ファイル名のセット

filename 1 は5CHから，filename 2 は6CHから格納されるので，6CH以後のデータをFCB 2に転送します(5CHからのファイルをオープンすると，6CH以後のデータはオーバーライトされて消去されてしまいます)．

その後で，FCB 1をopen，FCB 2をcreateします．ファイルのオープン／作成（create)／クローズが正常でないとレジスタALに0FFHが返されるので，これによって各エラー・メッセージを表示するルーチンに分岐します．

# 10.3 / listコマンドの作成

●スペース・キーによるフリーズ（画面表示の一時停止），リターン・キーによる表示速度をゆっくりにする機能を持つアスキー・ファイル表示コマンドを作成する

## プログラム

```
; -------------------------------------
; *    L I S T   C O M M A N D        *
; *        esc: kill command          *
; *         cr : slow disp            *
; *      space: freeze                *
; *     A>LIST filename               *
; -------------------------------------

INCLUDE B:STDIO.H
INCLUDE B:FILEH.H
TDELAY  EQU     2000H

CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,SS:CODE
        ORG     80H
FNAME   DB      128 DUP(?)          ; filename buffer

START:  MOV     BL,FNAME            ; ファイル名の終りに0を置く
        XOR     BH,BH
        MOV     BYTE PTR FNAME[BX+1],0
        openh   FNAME+2,0           ; open file
        JC      FIN
        MOV     HANDLE,AX           ; filehandle store

NEXT:   readh   DSKBUF,1024,HANDLE
        CMP     AX,0                ; ファイルの終り
        JE      FIN
        MOV     CX,AX               ; リード・データの表示
        MOV     BX,OFFSET DSKBUF
DISP:   CMP     BYTE PTR [BX],1AH   ; text end mark
        JE      FIN
ABC:    inkey                                        ①
        CMP     AL,ESC              ; esc key        ②
        JE      FIN
        CMP     AL,20H              ; space key
        JNE     BR1
        NOT     SPFLAG                               ③
BR1:    CMP     SPFLAG,0
        JNE     ABC
        CMP     AL,CR               ; cr key
        JNE     BR2
        NOT     CRFLAG                               ④
BR2:    CMP     CRFLAG,0
        JE      BR3
        PUSH    CX                  ; time delay
        MOV     CX,TDELAY                            ⑤
A1:     LOOP    A1
        POP     CX
BR3:    putchar [BX]
        INC     BX
        LOOP    DISP
        JMP     NEXT
FIN:    closeh  HANDLE              ; close file
        retms

HANDLE  DW      ?                   ; ファイル・ハンドル
DSKBUF  DB      1024 DUP(?)         ; リード・バッファ
```

```
SPFLAG  DB      0               ; space flag
CRFLAG  DB      0               ; cr flag

CODE    ENDS
        END     START
```

注）EXE2BINによりCOMファイルに変換すること．

CTRL＋Sキーによるフリーズ（画面表示の一時停止）は，同時に2つのキーを押さなければならず，少々不便なことがあります．このプログラムではスペース・キーによりフリーズ機能が使えるようにしました．また，リストの表示速度はかなり速いので，リターン・キーを押して表示速度を遅くできるようにしました．

これらは基本的には第8章8.10のTYPEコマンドにキー入力処理を付け加えたものです．

スペース・キー，リターン・キーによる各動作は，**トグル動作**（同じスイッチを一度押すとONになり，二度目でOFFになるようなもの）としたので，これらのキーが押されるたびにSPFLAGとCRFLAGのデータを論理演算子NOTによりビット反転し，0と－1の2値論理で扱います．

プログラムの中の注釈①～⑤は次の通りです．

①キー入力

②ESCキーが押されたならば，プログラムの実行を中止する

③スペース・キーが押されたらSPFLAGを反転し，CRFLAGが0でないならばフリーズする

④タイム・ディレイ．TDELAYの値を変えるとスロー動作の速度が変わる

# 10.4／llistコマンドの作成

●プリンタへのアスキー・ファイル表示コマンドを作成する

## プログラム

```
; -------------------------------------
; *    L L I S T   C O M M A N D      *
; *       A>LLIST filename            *
; -------------------------------------

INCLUDE B:STDIO.H
INCLUDE B:FILEH.H
```

```
CODE      SEGMENT
          ASSUME    CS:CODE,DS:CODE,SS:CODE
          ORG       80H
FNAME     DB        128 DUP(?)          ; filename buffer

START:    MOV       BL,FNAME            ; ファイル名の終りに0を置く
          XOR       BH,BH
          MOV       BYTE PTR FNAME[BX+1],0
          openh     FNAME+2,0           ; open file
          JC        FIN
          MOV       HANDLE,AX           ; filehandle store

          lprint    TABSET
NEXT:     readh     DSKBUF,1024,HANDLE
          CMP       AX,0                ; ファイルの終り
          JE        FIN
          MOV       CX,AX               ; プリンタ出力
          MOV       BX,OFFSET DSKBUF
DISP:     CMP       BYTE PTR [BX],1AH     ; text end mark
          JE        FIN
          lptout    [BX]
          INC       BX
          LOOP      DISP
          JMP       NEXT
FIN:      closeh    HANDLE              ; close file
          retms

HANDLE    DW        ?                   ; ファイル・ハンドル
DSKBUF    DB        1024 DUP(?)         ; リード・バッファ
TABSET    DB        ESC,'(009,017,025,033,041,049,057,065,073.$'

CODE      ENDS
          END       START
```

注）EXE2BIN により COM ファイルに変換すること.

MS-DOS では，CTRL+P または I／O リダイレクトにより，ファイルの内容をプリンタへ出力することができますが，ここではプリンタへの出力コマンド llist を作成します.

基本的には第８章8.10 の putchar を lptout（第８章の stdio. h 参照）に変更すればよいわけですが，問題が１つだけあります．それは，タブ（tab）に関するもので，プリンタにタブ位置を設定しておかないと，プリンタがタブ・コードを受け取ってもその動作を行わないのです.

そこで，まず次のようにエスケープ・シーケンス・コードをプリンタに送り，タブ設定を行わなければなりません.

```
ESC,‘（009,017,025…    .’
       ↑                 ↑
 3行でタブ位置を指定    最後はピリオド
```

なお，このコードはNEC PC-PR201用のものです．プリンタのタイプによっては異なるものもあるので，他のプリンタで利用する場合はそのプリンタのマニュアルを参照してください．

# 10.5／fdumpコマンドの作成

●アスキー・ファイル以外のファイルに関しても表示できるファイル・ダンプ・コマンドを作成する

**プログラム**

```
; --------------------------------------
; *   ファイル・ダンプ・プログラム       *
; *     A>FDUMP filename                *
; --------------------------------------

INCLUDE B:STDIO.H
INCLUDE B:FILE.H
INCLUDE B:MYLIB.H

CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,SS:CODE
        ORG     5CH
FCB     DB      34 DUP(?)
WORK    DW      ?
BUFF    DB      128 DUP(?)

START:  open    FCB             ; ファイル・オープン
        CMP     AL,0FFH         ; ファイルが見つからない
        JNE     NORMAL
        print   MSG             ; エラー・メッセージ
        retms
NORMAL: MOV     FCB+32,0        ; カレント・レコード・フィールドの設定
NEXT:   read    FCB
        CMP     AL,1
        JE      FIN
        dump    BUFF,8,WORK
        JMP     NEXT
FIN:    retms

MSG     DB      'file not found$'

CODE    ENDS
        END     START
```

注）EXE2BINによりCOMファイルに変換すること．

コマンドは,

A＞FDUMP filename ⏎

というように使用します.

これによってFCBにセットされたfilenameで示されるファイルをオープン(open FCB)し, 1レコード(128バイト) をDTAに読み込みます.

このデータをマクロdump(本章10.4参照)により16進ダンプします. このようすを図10.2に示します.

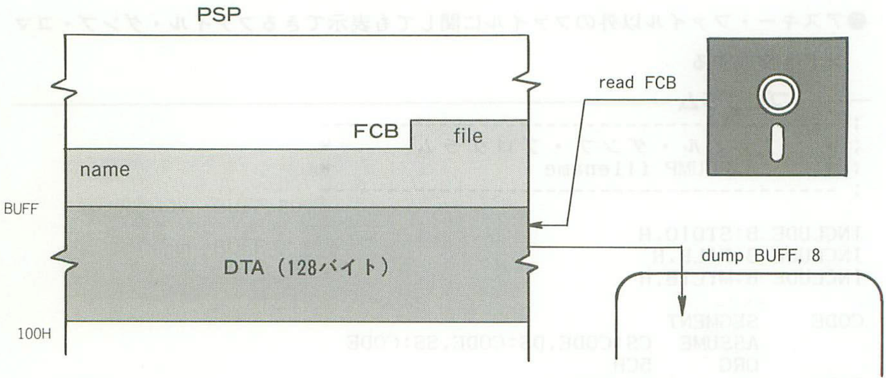



図10.2 ファイル・ダンプの概念

fdumpコマンドによる結果を図10.3に示します.

▼図10.3 fdumpの実行結果

```
A>B:FDUMP EX3.COM
B4 09 BA 0B 01 CD 21 B4 4C CD 21 0D 0A 48 65 6C ......!.L.!..Hel
6C 6F 20 4D 53 2D 44 4F 53 20 77 6F 72 6C 64 0D lo MS-DOS world.
0A 24 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .$..............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

# 10.6 ／ secdumpコマンドの作成

●ディスクから1セクタ分読み込み，16進ダンプするコマンドを作成する

## プログラム

```
; ----------------------------------------
; *   セクタ・ダンプ（DRIVE  A）       *
; *           A>SECDUMP                  *
; ----------------------------------------

INCLUDE B:STDIO.H
INCLUDE B:DSKRW.H
INCLUDE B:MYLIB.H

SECBYTE EQU      512                 ; 5"2DD のセクタサイズ
                                     ; if 8"2D then 1024
        dseg
MSG     DB       CR,LF,'sector number xxxx ?$'
INBUF   DB       5,6 DUP (?)
SEC     DW       ?
WORK    DW       ?
DISKBUF DB       SECBYTE DUP(?)
        dend

        sseg                         ; stack segment

        pseg
START:  MOV      AX,DATA             ; DS set
        MOV      DS,AX
NEXT:   print    MSG                 ; 相対セクタ番号の入力
        input    INBUF
        asc_hex  INBUF+4             ; 4桁のASCIIを2桁の16進に
        MOV      BYTE PTR SEC,AL
        asc_hex  INBUF+2
        MOV      BYTE PTR SEC+1,AL
        dskrd    0,DISKBUF,1,SEC ; DRIVE  Aから1セクタ読む
        dump     DISKBUF,SECBYTE/16,WORK
        JMP      NEXT
        pend

        END      START
```

マクロinputにより入力した4桁のアスキー・コードを上位桁からINBUF＋2に格納します．これをasc_hexにより16進数に変換し，下位バイト，上位バイトの順でSECに格納します．

ドライブA（0）のSECで示される相対セクタ番号から1セクタ分DISK-

BUFにアブソリュート（ダイレクト）・ディスク・リードを行います．1セクタのサイズ（フロッピィディスクの媒体によって異なる）はSECBYTEによって定義されているのでDISKBUFからSECBYTE分の内容をマクロdumpでスクリーンに表示します．パラメータのWORKは，マクロ内で16進→アスキー・コード変換する際に使用する作業領域です．

以下の動作の概要を図10.4に示します．

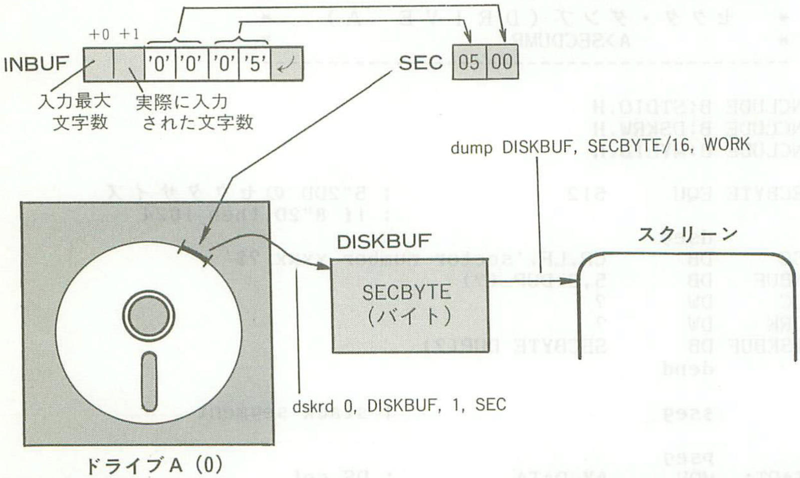



図10.4　セクタ・ダンプの概念

セクタ・ダンプsecdumpの実行結果を図10.5に示します．

▼secdumpの実行結果

```
A>SECDUMP

sector number xxxx ?0005
50 43 39 38 30 30 5F 53 59 53 20 08 00 00 00 00 PC9800_SYS .....
00 00 00 00 00 00 BA 86 34 07 00 00 00 00 00 00 .........4.......
49 4F 20 20 20 20 20 20 53 59 53 27 00 00 00 00 IO      SYS'....
00 00 00 00 00 00 5B 7E 31 07 02 00 00 A0 00 00 ......[~1.......
4D 53 44 4F 53 20 20 20 53 59 53 27 00 00 00 00 MSDOS   SYS'....
00 00 00 00 00 00 40 72 33 07 2A 00 00 45 00 00 ......@r3.*..E..
43 4F 4D 4D 41 4E 44 20 43 4F 4D 20 00 00 00 00 COMMAND COM ....
00 00 00 00 00 00 6E 82 33 07 3C 00 26 43 00 00 ......n.3.<.&C..
4B 4E 4A 44 49 43 20 20 53 59 53 20 00 00 00 00 KNJDIC  SYS ....
00 00 00 00 00 00 CA 72 34 07 4D 00 00 DC 00 00 .......r4.M.....
43 48 4B 44 53 4B 20 20 43 4F 4D 20 00 00 00 00 CHKDSK  COM ....
00 00 00 00 00 00 B5 BB 24 07 84 00 76 1A 00 00 .........$...v...
44 45 42 55 47 20 20 20 43 4F 4D 20 00 00 00 00 DEBUG   COM ....
00 00 00 00 00 00 6D 04 25 07 8B 00 72 2F 00 00 ......m.%...r/..
45 44 4C 49 4E 20 20 20 43 4F 4D 20 00 00 00 00 EDLIN   COM ....
```

```
00 00 00 00 00 00 34 67 31 07 97 00 A4 23 00 00 ......4g1....#..
4D 4F 52 45 20 20 20 20 43 4F 4D 20 00 00 00 00 MORE    COM ....
00 00 00 00 00 00 82 BD 24 07 A0 00 11 11 00 00 ........$.......
52 45 43 4F 56 45 52 20 43 4F 4D 20 00 00 00 00 RECOVER COM ....
00 00 00 00 00 00 7D BE 24 07 A5 00 2F 0A 00 00 ......}.$.../...
53 59 53 20 20 20 20 20 43 4F 4D 20 00 00 00 00 SYS     COM ....
00 00 00 00 00 00 2D BF 24 07 A8 00 C8 03 00 00 ......-.$.......
43 52 45 46 20 20 20 20 45 58 45 20 00 00 00 00 CREF    EXE ....
00 00 00 00 00 00 00 80 5E 05 A9 00 00 36 00 00 ........^....6..
45 58 45 32 42 49 4E 20 45 58 45 20 00 00 00 00 EXE2BIN EXE ....
00 00 00 00 00 00 CC A9 24 07 B7 00 78 06 00 00 ........$...x...
46 43 20 20 20 20 20 20 45 58 45 20 00 00 00 00 FC      EXE ....
00 00 00 00 00 00 01 BD 24 07 B9 00 52 0A 00 00 ........$...R...
46 49 4E 44 20 20 20 20 45 58 45 20 00 00 00 00 FIND    EXE ....
00 00 00 00 00 00 37 BD 24 07 BC 00 1E 19 00 00 ......7.$.......
4C 49 42 20 20 20 20 20 45 58 45 20 00 00 00 00 LIB     EXE ....
00 00 00 00 00 00 00 80 5E 05 C3 00 80 7D 00 00 ........^....}..

sector number xxxx ?^C
```

# 10.7／atrsetコマンドの作成

●ファイル属性のうち，リード・オンリー属性とリード／ライト属性に関する設定を行うコマンドを作成する

## プログラム

```
; -------------------------------------------
; *    ファイル属性の設定プログラム         *
; *        A>ATRSET filename{/R/W}          *
; -------------------------------------------

CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,SS:CODE
        ORG     80H
FNAME   DB      128 DUP(?)

START:  MOV     BL,FNAME          ; ファイル名の終りに0を置く ①
        XOR     BH,BH
        MOV     BYTE PTR FNAME[BX-1],0
        MOV     AH,43H            ; 属性を得る ②
        MOV     DX,OFFSET FNAME+2
        MOV     AL,0
        INT     21H
        CMP     FNAME[BX],'R'     ; read only 属性 ③
        JNE     BR1
        OR      CX,1              ; ビット0をセット
BR1:    CMP     FNAME[BX],'W'     ; r/w 属性
        JNE     BR2
        AND     CX,0FFFEH         ; ビット0をリセット
BR2:    MOV     AH,43H            ; 属性の設定 ④
        MOV     DX,OFFSET FNAME+2
```

```
        MOV     AL,1
        INT     21H
        MOV     AH,4CH              ; return MS-DOS   ④
        INT     21H
CODE    ENDS
        END     START
```

注）EXE2BINによりCOMファイルに変換すること.

ファイル属性の取得／設定はファンクション43Hにより行いますがレジスタALに0を与えると取得モード，1を与えると設定モードになります．ファイルの属性のセットはレジスタCXに与えます．

このコマンドは,

A＞ATRSET　filename／［RまたはW］

注）RまたはWは大文字で与えること

のように使用し,／の後がRならばfilenameで示されるファイルの属性をリード・オンリー属性，Wならばリード／ライト属性に設定します.

コマンド・ラインのパラメータは図**10.6**のようにPSPに渡されます.

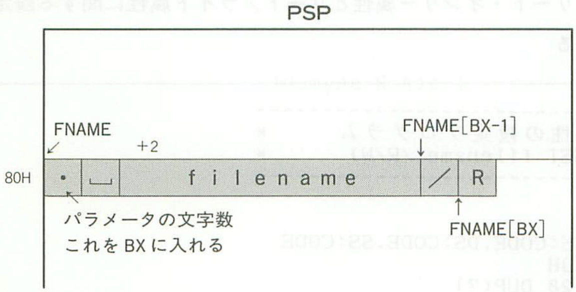



図10.6　PSPへのコマンドラインの格納

プログラムの中の注釈①～⑤は次の通りです.

①ファイル名の先頭アドレスはFNAME＋2で，ファイル名の最後(図の／の位置)に0をセットし，UNIX型文字列を完成させる

②ファイルの属性を得る

③R／Wのスイッチの格納位置はFNAME[BX]で示される．このデータが'R'ならばファイルの属性のビット0を1にセットし，'W'ならばファイルの属性のビット0を0にリセットします．ファイルの属性のビットは第7章**7.4**を参考のこと

④ファイルの属性の決定

# 10.8 ／ gettimeコマンドの作成

●MS-DOSの内部時計から時刻を得て表示するコマンドを作成する

**プログラム**

```
; ----------------------------------
; *        時 刻 を 得 る           *
; *           A>GETTIME            *
; ----------------------------------

INCLUDE B:STDIO.H
INCLUDE B:MYLIB.H

GETTIME MACRO               ; gettime function
        MOV     AH,2CH
        INT     21H
        ENDM

        dseg
TIME    DB      ?,?,'時$'
MINUTE  DB      ?,?,'分$'
SECOND  DB      ?,?,'秒$'
        dend

        sseg                ; stack segment

        pseg
START:  MOV     AX,DATA
        MOV     DS,AX
        gettime             ; 時 刻 を 得 る
        PUSH    DX
        PUSH    CX
        hex_bcd CH,TIME              ; 時 の 表 示
        print   TIME
        POP     CX
        hex_bcd CL,MINUTE            ; 分 の 表 示
        print   MINUTE
        POP     DX
        hex_bcd DH,SECOND            ; 秒 の 表 示
        print   SECOND
        retms
        pend
        END     START
```

MS-DOSの内部時計から時，分，秒を得るには，ファンクション2CHを利用します．これによって次のレジスタに時刻のデータを得ることができます．

CH：時（0～23）

CL：分（0～59）

DH：秒（0～58）

これらのレジスタの内容をマクロ hex_bcd により 10 進アスキー文字に変換します．変換結果はそれぞれ TIME, MINUTE, SECOND にセットします．これらのフィールドにはあらかじめ‘時’，‘分’，‘秒’という文字をセットしておきます．実行結果は次のようになります．

```
A>B:GETTIME
19時43分39秒
```

## 10.9 ／ボール反射プログラム

●スクリーン上に作った壁の中をボールが反射しながら飛び回るプログラムを作成する

### プログラム

```
;----------------------------------------
;*          ボール反射プログラム          *
;----------------------------------------

INCLUDE B:STDIO.H
INCLUDE B:SCREEN.H

        dseg
        escdat
CURSOFF DB      ESC,'[>5h$'
X       DB      7       ; ボールの X 座標
Y       DB      10      ; ボールの Y 座標
ADX     DB      1       ; X の増分
ADY     DB      1       ; Y の増分
WALL1   DB      '##############################################',CR,LF,'$'
WALL2   DB      ']                                            [',CR,LF,'$'
        dend

        sseg            ; stack segment

        pseg
START:  MOV     AX,DATA
        MOV     DS,AX
        cls
        color   5
        print   CURSOFF ; cursole off
        print   WALL1   ; 壁の表示
        MOV     CX,14
```

```
ABC:     print   WALL2
         LOOP    ABC
         print   WALL1
NEXT:    locate  X,Y      ; ボール表示
         putchar 'o'
         MOV     CX,5000H ; time delay
DELAY:   LOOP    DELAY
         CMP     X,2
         JE      INVAX    ; 縦壁のチェック
         CMP     X,44
         JE      INVAX
YCHECK:  CMP     Y,2      ; 横壁のチェック
         JE      INVAY
         CMP     Y,15
         JE      INVAY
VANISH:  locate  X,Y      ; ボールを消す
         putchar ' '
         MOV     AL,ADX   ; ボールを進める
         ADD     X,AL
         MOV     AL,ADY
         ADD     Y,AL
         JMP     NEXT
INVAX:   NEG     ADX      ; X方向の増分反転
         JMP     YCHECK
INVAY:   NEG     ADY      ; Y方向の増分反転
         JMP     VANISH
         pend
         END     START
```

これはユーティリティではありませんが，CRT スクリーン上で動きのあるプログラムを作る際の参考として例示しました．

ボールの移動原理を**図 10.7** に示します．

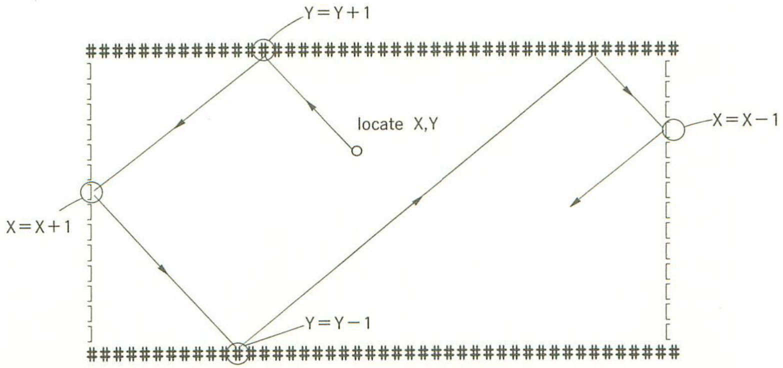



図10.7　ボールの移動原理

ボールの位置はマクロlocate X, Yで示されます．ボール位置を進める式は，

```
X = X + ADX
Y = Y + ADY
```

となり，横壁(上下の壁)に当たるとADYの符号を反転させ，縦壁(左右の壁)に当たるとADXの符号を反転させます．したがって，ADX，ADYの内容は，1と−1のトグル動作となります．

一般にCRTスクリーンはカーソルONモードであるため，このままではボール移動中にカーソルが点滅しているので見辛いものになります．そこでカーソルOFFモードにするために次のエスケープ・シーケンス

```
ESC, '[>5h'
```

を送ります．

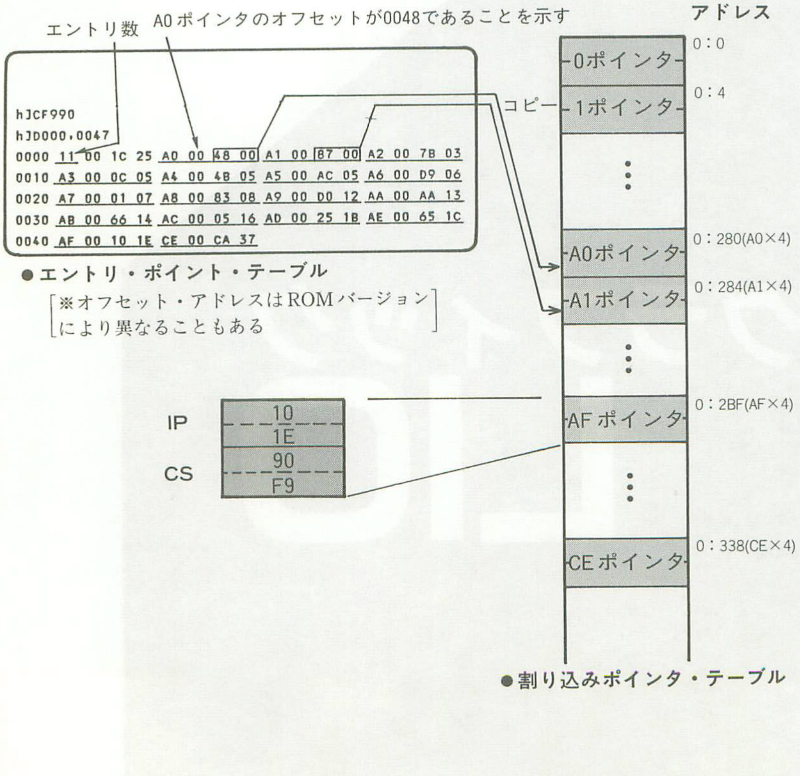
# 第11章
# グラフィック・インターフェース

グラフィック
LIO

## 本章の目的

MS-DOS や CP／M-86 などのパソコン用 OS はグラフィック機能をサポートしてはいません．これらの DOS 上からグラフィック処理を行うためには，各パソコンのハードウェアに密着したインターフェースが必要になります．ここでは，PC-9801用のグラフィック・インターフェースとしてグラフ LIO を解説し，これを用いて screen，pset，line，circle などの機能を作ることにします．さらに，マウスを用いたグラフィック処理の例も示します．

▼図11.1　グラフ LIO 用割り込みベクトルのセット



# 11.1 ／ グラフLIO*（Logical Input Output）

PC-9800 シリーズにおいてグラフィック処理を行うには，専用のソフトウェア割り込みを使って，簡単に ROM 内ルーチンを利用することができます．

グラフ LIO のソフトウェア割り込みのコードは A 0～AF，CE の 17 種類あり，レジスタ BX にパラメータ・リスト（各種座標データなど）の先頭アドレス（オフセット）を与えて，INT nn を行えば目的のグラフィック処理が行えます．

### ●割り込みベクタの設定

グラフ LIO による割り込みを行うためには，アドレス 0：280～0：2 BF および 0：338 の割り込みポインタ・テーブルに LIO 処理ルーチンへのエントリ・ポイントがセットされていなければなりません．

このエントリ・ポイントは ROM の F 990： 0 以後に格納されているので，ユーザは，それを 0 ：280 以後の割り込みポインタ・テーブルにコピーすればよいことになります．このようすを**図 11.1** に示します．コピーに際しては，各ポインタのオフセット・アドレス（0048，0087…）をメモリの下位に，セグメント・アドレス（F 990：固定）をメモリの上位に置くものとします．

### ●グラフ LIO のコマンド

グラフ LIO で利用できるコマンドは**表 11.1** に示すように 17 種類あります．グラフ LIO のコマンドを実行するには，レジスタ BX にパラメータ・リストの先頭アドレス（オフセット）を与えて次のようにソフトウェア割り込みを行います．nn は割り込み番号（A 0～AF，CE）を表しています．

```
MOV   BX,パラメータ・リストの先頭アドレス（オフセット）
INT   nn
```

＊グラフ LIO 『PC-9800 グラフ LIO』（インターフェース，1983年9月号）
『PC-98 用拡張システムコール詳説』（インターフェース，1985年4月号別冊）
どちらも，白田耕作 CQ 出版社

●表11.1　グラフLIOコマンド

| コマンド名 | ソフトウェア割り込み番号 | 機　能 | パラメータ |
|---|---|---|---|
| INIT | A0 | グラフLIOの初期化 | ―― |
| SCREEN | A1 | グラフィック画面のモード設定 | +0 画面モード / +1 画面スイッチ / +2 アクティブ画面 / +3 ディスプレイ画面 |
| VIEW | A2 | ビューポートの指定 | +0 $x_1$ / +2 $y_1$ / +4 $x_2$ / +6 $y_2$ / +8 領域色 / +9 境界色 |
| COLOR1 | A3 | カラー指定 | +0 未使用 / +1 バックグラウンド / +2 ボーダー / +3 フォアグラウンド |
| COLOR2 | A4 | パレット指定 | +0 パレット番号 / +1 カラーコード |
| CLS | A5 | 画面クリア | ―― |
| PSET | A6 | 指定点に点をセット | +0 x / +2 y / +4 パレット番号 |
| LINE | A7 | 指定点を直線で結ぶ | +0 $x_1$ / +2 $y_1$ / +4 $x_2$ / +6 $y_2$ / +8 パレット / +9 画面コード / +10 スイッチ / +11 ラインスタイル |
| CIRCLE | A8 | 円を描く | 中心: +0 cx / +2 cy; 半径: +4 rx / +6 ry; +8 パレット / +9 フラグ; 開始点: +10 sx / +12 sy; 終了点: +14 ex / +16 ey |
| PAINT1 | A9 | ぬりつぶし | +0 x / +2 y / +4 領域色 / 境界色 / ワークエリア: +6 end / +8 top |
| PAINT2 | AA | タイルパターンによるぬりつぶし | |
| GET | AB | 描面パターンの格納 | |

| | | | |
|---|---|---|---|
| PUT1 | AC | 描画パターンの表示 | 説明略，<br>詳細は，『PC-98用拡張システムコール詳説』<br>（インターフェース，1985年4月号別冊・付録）<br>白田 耕作，CQ出版社<br>を参考にされたい．<br>また，LINEとCIRCLEのパラメータはPC-9801E/F/Mでは増えている．この点も上記文献を参考にされたい． |
| PUT2 | AD | 漢字の表示 | |
| ROLL | AE | 指定ドット数のロール・アップ | |
| POINT2 | AF | 指定位置のパレット番号を得る | |
| GCOPY | CE | 指定域のグラフィックデータをメモリ転送 | |

●グラフ LIO の作業領域

グラフ LIO ではデータ・セグメント（DS）のオフセット 0 ～1400 H の範囲を図 11.2 のような作業領域として使用します．

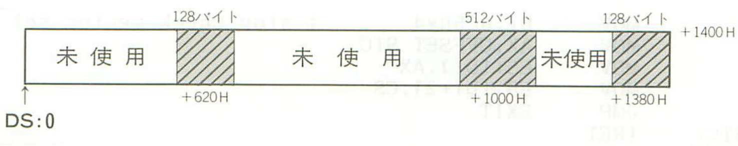



図11.2　グラフ LIO の作業領域

# 11.2 グラフLIOインターフェース・ライブラリの作成

グラフ LIO を利用するための各種設定やグラフィック・コマンドをマクロ定義したヘッダ・ファイル glib .h を**リスト 11.1** に示します.

●リスト11.1　glib.h

```
; -----------------------------------------------
; *        グラフＬＩＯ・ライブラリ            *
; -----------------------------------------------

GSET    MACRO               ; GLIO vector set
        LOCAL   ABC
        MOV     AX,0F990H        ; A0 - AF vector set   ┐
        MOV     DS,AX                                   │
        XOR     BX,BX                                   │
        MOV     ES,BX                                   │
        MOV     SI,6                                    │
        MOV     DI,0A0H*4                               │ ①
        MOV     CX,16                                   │
        CLD                                             │
ABC:    MOVSW                                           │
        STOSW                                           │
        ADD     SI,2                                    │
        LOOP    ABC                                     ┘

        MOV     DI,0CEH*4        ; CE vector set        ┐
        MOVSW                                           │ ②
        STOSW                                           ┘

        MOV     DI,0C5H*4        ; stop check vector set ┐
        MOV     AX,OFFSET STC                            │
        MOV     ES:[DI],AX                               │
        MOV     ES:[DI+2],CS                             │ ③
        JMP     EXIT                                     │
STC:    IRET                                             ┘
EXIT:
        ENDM

GINIT   MACRO               ; GLIO initialize
        INT     0A0H
        ENDM

SCREEN  MACRO                    ; screen 3,0
        MOV     BX,OFFSET S_ADR
        INT     0A1H
        ENDM

CLS     MACRO                    ; cls
        INT     0A5H
        ENDM
```

```
PSET     MACRO   X,Y,C            ; pset(x,y),c
         MOV     AX,X             ; x set
         MOV     PSET_X,AX
         MOV     AX,Y             ; y set
         MOV     PSET_Y,AX
         MOV     AL,C             ; color set
         MOV     PSET_C,AL
         MOV     BX,OFFSET PSET_X
         INT     0A6H
         ENDM

LINE     MACRO   X1,Y1,X2,Y2,C    ; line (x1,y1)-(x2,Y2),c
         MOV     AX,X1            ; x1 set
         MOV     LINE_X1,AX
         MOV     AX,Y1            ; y1 set
         MOV     LINE_Y1,AX
         MOV     AX,X2            ; x2 set
         MOV     LINE_X2,AX
         MOV     AX,Y2            ; y2 set
         MOV     LINE_Y2,AX
         MOV     AL,C             ; color set
         MOV     LINE_C,AL
         MOV     BX,OFFSET LINE_X1
         INT     0A7H
         ENDM

CIRCLE   MACRO   X,Y,R,C          ; circle (x,y),r,c
         MOV     AX,X             ; x set
         MOV     CIRC_X,AX
         MOV     AX,Y             ; y set
         MOV     CIRC_Y,AX
         MOV     AX,R             ; r set
         MOV     CIRC_RX,AX
         MOV     CIRC_RY,AX
         MOV     AL,C             ; color set
         MOV     CIRC_C,AL
         MOV     BX,OFFSET CIRC_X
         INT     0A8H
         ENDM

G_DAT    MACRO
S_ADR    DB      3         ; 画面モード
         DB      0         ; 画面スイッチ
         DB      0FFH      ; アクティブ画面
         DB      0FFH      ; ディスプレイ画面

PSET_X   DW      0         ; x座標
PSET_Y   DW      0         ; y座標
PSET_C   DB      7         ; パレット色

LINE_X1  DW      0         ; 始点x
LINE_Y1  DW      0         ; 始点y
LINE_X2  DW      0         ; 終点x
LINE_Y2  DW      0         ; 終点y
LINE_C   DB      7         ; パレット色
         DB      0         ; 描画指定（直線）
```

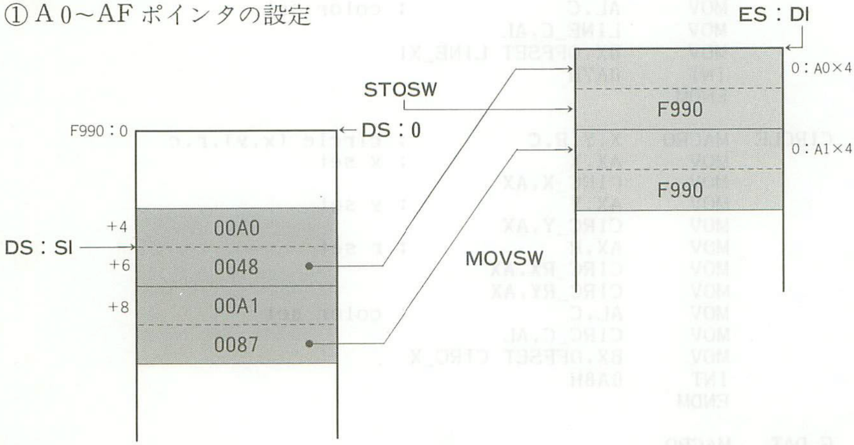
```
CIRC_X  DW      0       ; 中心 x
CIRC_Y  DW      0       ; 中心 y
CIRC_RX DW      0       ; 半径 r x
CIRC_RY DW      0       ; 半径 r y
CIRC_C  DB      7       ; パレット色
        DB      0       ; フラグ

        ORG     620H
        DB      0DE0H DUP(?)    ; GLIO作業領域
        ENDM
```

以下，各マクロの説明を行います．

## ● gset：グラフ LIO ベクタの設定

①A 0～AF ポインタの設定




プリミティブ・ストリング命令 MOVSW によりエントリ・ポイントの IP 値をポインタ・テーブルにコピーします．MOVSW 命令はレジスタ SI，DI を＋2します．

プリミティブ・ストリング命令 STOSW によりエントリ・ポイントの CS 値（レジスタ AX に格納されている F 990 H という値）をポインタ・テーブルにセットします．STOSW 命令はレジスタ DI のみを＋2するので，ADD SI, 2 によりレジスタ SI を＋2しておきます．

②CE ポインタの設定

③ストップ・チェック・ベクタの設定

circleやpaintなどの比較的，処理に手間がかかる機能はある処理単位毎にC5Hポインタへ INTします．ユーザはC5Hポインタにユーザ・サービス・ルーチンへのエントリ・ポイントを設定しておかなければなりません．

一般にユーザ・サービス・ルーチンには，STOPキーなどの受付け処理を置きますが，ここではユーザ・サービス・ルーチンSTCとして，IRETのみを置きました．これは単にINTに対して元へ戻るというだけの動作を行います．

### ● ginit：グラフ LIO の初期設定

グラフLIOの使用に際して，最初に1度だけ呼び出さなければならないマクロです．

### ● screen：グラフィック画面モードの設定

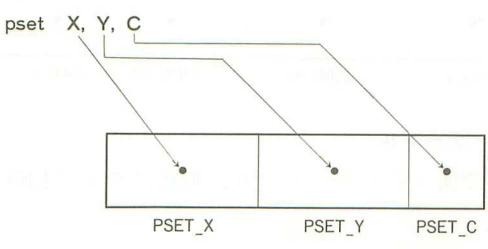
無条件にscreen 3，0を実行します．データ・テーブルのS_ADR以後にスクリーン・モードのデータがあらかじめ設定されています．パラメータを指定するscreenコマンドを作成したい場合には，pset, line, circleの作り方を参考にしてください．

### ● cls：グラフィック画面の消去

アクティブ画面のビューポート内をバック・グラウンド・カラーで塗りつぶします．

### ● pset：点のセット

下図のようなマクロコールにより，(X，Y)の位置にパレット色Cの点をセットします．パラメータX, Y, Cにはレジスタでもメモリでも記述できるよう



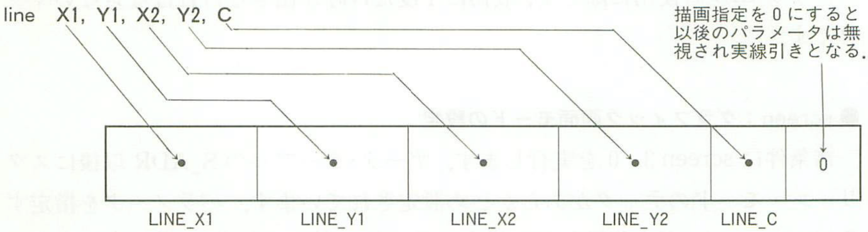

にレジスタAXを介してメモリへパラメータを渡す方式にしました．これは，

lineやcircleでも同じです．

● line：直線を引く

下図のようなマクロコールにより，(X 1，Y 1)－(X 2，Y 2) の位置にパレット色Cの直線が引けます．

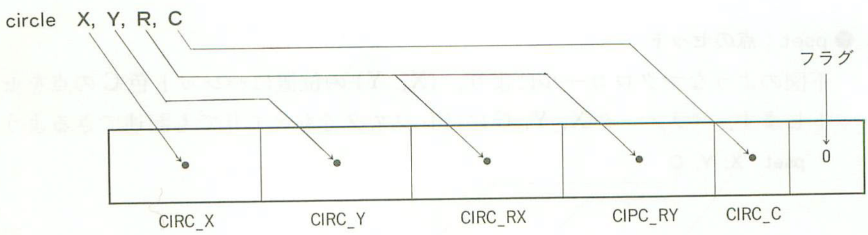
描画指定を0に固定してあるので，このマクロlineは直線しか引くことができません．この描画指定を1にすればbox，2にすればboxfillの指定となります．




● circle：円の描画

下図のようなマクロコールにより，(X，Y) を中心とする半径Rの真円をパレット色Cで描きます．

半径RX，RYに同じ値Rを入れるようにしているので，楕円は描けません．フラグを0にしたので以後のパラメータは無視します．




● g_dat：グラフLIOのデータ・テーブル

screen，pset，line，circleで使うデータ・テーブル．最後にグラフLIO用の作業領域を確保してあります．

# 11.3／psetの利用例

psetを用いて，斜めの点線を描く

## プログラム

```
; ----------------------------------------
; *      p s e t   e x a m p l e         *
; ----------------------------------------

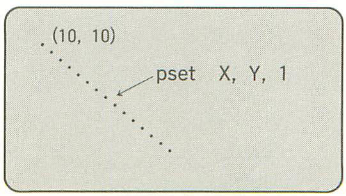
INCLUDE B:STDIO.H
INCLUDE B:GLIB.H

        dseg
X       DW      10      ; x 座標
Y       DW      10      ; y 座標
        g_dat           ; GLIO data table
        dend

        sseg            ; stack segment

        pseg
START:  gset            ; GLIO vector set
        MOV     AX,DATA
        MOV     DS,AX
        ginit           ; GLIO initialize
        screen          ; screen 3.0
        cls
        MOV     CX,50
DRAW:   PUSH    CX
        pset    X,Y,1   ; pset(x,y),1
        ADD     X,5
        ADD     Y,5
        POP     CX
        LOOP    DRAW
        retms           ; return MS-DOS
        pend
        END     START
```

座標(10, 10)の点からX, Yを5ずつ増加させながら，pset X,Y,1を行うことにより，青色の点線を斜めに引きます．グラフLIOへの割り込みにより，CS, DS, SS, IP, SP以外のレジスタは破壊されるので，保存しなければならないレジスタはユーザの責任でPUSH／POPを行います．



# 11.4 / lineの利用例

line を用いて，20 本の縦線を描く

## プログラム

```
; -------------------------------------------
; *      l i n e   e x a m p l e          *
; -------------------------------------------

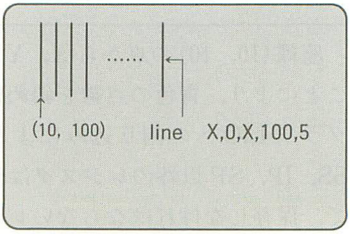
INCLUDE B:STDIO.H
INCLUDE B:GLIB.H

        dseg
X       DW      10      ; x 座標
        g_dat           ; GLIO data table
        dend

        sseg            ; stack segment

        pseg
START:  gset            ; GLIO vector set
        MOV     AX,DATA
        MOV     DS,AX
        ginit           ; GLIO initialize
        screen          ; screen 3,0
        cls
        MOV     CX,20
NEXT:   PUSH    CX
        line    X,0,X,100,5     ; line (x,0)-(x,100),5
        ADD     X,10
        POP     CX
        LOOP    NEXT
        retms                   ; return MS-DOS
        pend
        END     START
```

Xの値を10から10ずつ飛ばして20回変化させ，そのたびに直線を水色で引きます．



# 11.5 ／circleの利用例

circle を用いて，中心（200，100）の同心円を色を変えて描く

## プログラム

```
; ----------------------------------------
; *    c i r c l e   e x a m p l e       *
; ----------------------------------------

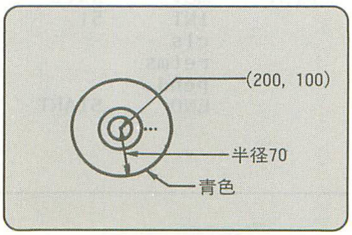
INCLUDE B:STDIO.H
INCLUDE B:GLIB.H

        dseg
R       DW      10          ; r 座標
        g_dat               ; GLIO data table
        dend

        sseg                ; stack segment

        pseg
START:  gset                ; GLIO vector set
        MOV     AX,DATA
        MOV     DS,AX
        ginit               ; GLIO initialize
        screen              ; screen 3,0
        cls
        MOV     CX,7
DRAW:   PUSH    CX
        circle  200,100,R,CL     ; circle(200,100),R,CL
        ADD     R,10
        POP     CX
        LOOP    DRAW
        retms                    ; return MS-DOS
        pend
        END     START
```

半径Rを10から10ずつ飛ばして増加させます．円の色はループ回数CXの下位バイトCLを用いて7，6,…,1と変化させます．



# 11.6 / グラフィックスへのマウスの利用例

マウスのカーソルの位置に点を描き，マウスの移動した軌跡をグラフィック画面に描く

## プログラム

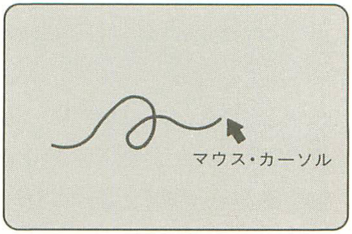
```
; -----------------------------------
; *     マウス  e x a m p l e      *
; -----------------------------------

INCLUDE B:STDIO.H
INCLUDE B:GLIB.H

        dseg
        g_dat           ; GLIO data table
        dend

        sseg            ; stack segment

        pseg
START:  gset            ; GLIO vector set
        MOV     AX.DATA
        MOV     DS,AX
        ginit           ; GLIO initialize
        screen          ; screen 3,0
        cls
        MOV     AX.18           ; カーソル色
        MOV     BX,2            ; 0:青 1:赤 2:緑
        INT     51
        MOV     AX,1            ; カーソルの表示
        INT     51
DRAW:   MOV     AX,3            ; カーソル位置の取得
        INT     51              ; CX <-- X   DX <-- Y
        CMP     AX,0            ; push botton
        JNE     FIN
        pset    CX,DX,<4 XOR 5>
        JMP     DRAW
FIN:    MOV     AX,2            ; カーソルを消す
        INT     51
        cls
        retms
        pend
        END     START
```




①マウスの位置がレジスタCX（X座標），DX（Y座標）に求められる
②マウスの左ボタンを押すとFINへ飛ぶ
③①で求められた位置をpsetでセットする．色を4 XOR 5としてあるのは，マウス・カーソルの色が4であり，同じ色が重なることで見えなくならないための処置．

ここで使用したマウスはPC-9801用のPC-9871です．マウスについての詳しい説明は98 FANBOOKS ①『CP／M-86 vsMS-DOS』（河西朝雄・著／技術評論社・刊）を参考にしてください．

# 第12章

# MS-DOS ver 3.1のための補足事項

MS-DOS
ver 3.1

## 本章の目的

これまで本書では，MS-DOS ver 2.0 上でのマクロアセンブラ MASM によるアセンブリ言語プログラムの開発を説明してきましたが，最近 **MS-DOS ver 3.1** がリリースされました．

ver 2.0 から ver 3.1 の主な変更点は，ネットワーク関係のサポートであり，それに関するコマンドやシステムコールが追加されたことです．

しかし，マクロアセンブラでプログラムを開発していく基本的な方法は，ver 2.0 も ver 3.1 も全く変わらず，本書のプログラムはそのまま ver 3.1 でも使えます．したがって，マクロアセンブラの開発環境としての MS-DOS ver 3.1 は，MASM，LINK，LIB，CREF などのユーティリティのバグの修正やスイッチ類の追加程度で，大きな変更点はありません．

ただ，DEBUG の代わりに**シンボリック・デバッガ SYMDEB**，UNIX ライクな**プログラム・メインテナ MAKE** などが提供され，プログラム開発がより効率的に行えるようになっています．

そこで，ここでは，MS-DOS ver 3.1 におけるマクロアセンブラ開発に関する追加・補足事項を説明します．

# 12.1／シンボリック・デバッカ(SYMDEB)

● SYMDEB の概要

MS-DOS ver 3.1では，今までのデバッガ DEBUGの代わりに，**シンボリック・デバッガ（SYMDEB）**が提供されています．

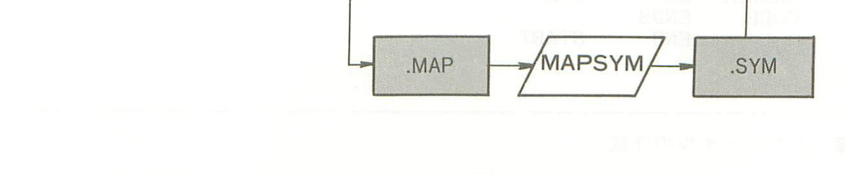
シンボリック・デバッガとは，その名が示すように，デバッグ時のサブコマンドのパラメータにアドレスではなく，**シンボル（名前）**を用いることができるデバッガです．

シンボル名を用いてデバッグを行うためには次の作業を行います．

・LINK時に**シンボル・マップ・ファイル(ファイルのタイプは .MAP)**を作成
・ユーティリティMAPSYMを用いて，.MAPファイルからデバッガSYMDEBに入力できる .SYMファイルを作成
・.SYM ファイルと被テスト・プログラム（.EXE または.COM）を SYMDEBにロードし，デバッグを行う

以上の概要を**図 12.1** に示します．

図12.1　SYMDEBの運用形態



それでは，実際にSYMDEBを用いてシンボリック・デバッグを行ってみることにします．デバッグを行うソース・プログラムの例は**図12.2**に示します．

●ソース・テキスト

デバッグ時に使用したいシンボル名は，PUBLIC宣言しておきます．

シンボルがPUBLIC宣言されていないと，.MAPファイルにシンボル名に関する情報が入りません．

▼図12.2　ソース・テキスト例

```
; --------------------------------------
; *     １０進のアスキーコードに変換      *
; --------------------------------------

CODE      SEGMENT
          ASSUME    CS:CODE,DS:CODE,SS:CODE
          PUBLIC    START,DCMAL,ABC,RESULT    ←シンボリック．デバッグ時に使うシンボル名は，PUBLIC宣言しておく
          ORG       100H
START:    MOV       BX,OFFSET RESULT
          MOV       AL,DATA
          MOV       CL,100    ; 第３桁
          CALL      DCMAL
          MOV       CL,10     ; 第２桁
          CALL      DCMAL
          MOV       CL,1      ; 第１桁
          CALL      DCMAL
          MOV       AH,4CH    ; return MS-DOS
          INT       21H

DCMAL:    MOV       DL,2FH    ; サブルーチン
ABC:      INC       DL
          SUB       AL,CL
          JNC       ABC
          MOV       [BX],DL
          INC       BX
          ADD       AL,CL
          RET

DATA      DB        231
RESULT    DB        3 DUP(?)
CODE      ENDS
          END       START
```

● .MAPファイルの作成

LINK時に.MAPファイルの指定を行います．スイッチ／MAPにより，シンボル名に関する情報が.MAPに入ります．

▼図12.3 .MAPファイルの作成

```
A>LINK

Microsoft 8086 Object Linker
Version 3.00 (C) Copyright Microsoft Corp 1983, 1984, 1985

Object Modules [.OBJ]: B:CHECK
Run File [A:CHECK.EXE]: B:CHECK
List File [NUL.MAP]: B:CHECK/MAP ←―――――.MAPファイルの作成
Libraries [.LIB]:
Warning: no stack segment
```

●.SYM ファイルの作成

ユーティリティMAPSYMを用いて，.MAPファイルを .SYMに変換します，

▼図12.4 .SYM ファイルの作成

```
A>MAPSYM B:CHECK.MAP
Microsoft Symbol File Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
        Program entry point at 0000:0100
```

● SYMDEB によるデバッグ

図 12.2 のソース・プログラムは， .COM モデルなので，SYMDEB には .SYM と .COM を入力します．

シンボル名は次のような書式で入力できます．

　[シンボルが定義されているセグメント名：] シンボル名 [±n]

セグメント名としては CS，DS，SS，ES などを書くことができます．±n はバイト数を示します．

▼図12.5 SYMDEB によるデバッグ

```
A>SYMDEB B:CHECK.SYM B:CHECK.COM ←――――――SYMDEBの起動
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-R
AX=0000  BX=0000  CX=002B  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1A86  ES=1A86  SS=1A86  CS=1A86  IP=0100   NV UP EI PL NZ NA PO NC
1A86:0100 BB2801         MOV    BX,RESULT
-G=CS:START,CS:DCMAL ←――――――シンボル名START, DCMAL
AX=00E7  BX=0128  CX=0064  DX=0000  SP=FFFC  BP=0000  SI=0000  DI=0000
```

```
DS=1A86  ES=1A86  SS=1A86  CS=1A86  IP=0119   NV UP EI PL NZ NA PO NC
1A86:0119 B22F          MOV     DL,2F                          :'/'
-G

Program terminated normally (0)
-DB DS:RESULT,RESULT+10 ←―――――――――――――――――シンボル名RESULT
1A86:0120                         -32 33 31 00 00 00 00 00      231.....
1A86:0130  00 00 00 00 00 00 00 00-00                       .........
-
-
-
-
```

一般にSYMDEBの起動法は次の3つがあります.

A＞SYMDEB

A＞SYMDEB　被デバッグ・ファイル（.EXE または.COM）

A＞SYMDEB　SYM ファイル　被デバッグ・ファイル

## 12.2 SYMDEBとDEBUGにおける サブコマンドの比較

SYMDEBのサブコマンドは，DEBUGのサブコマンドに対しては上位コンパチブルであると考えて構いません.各サブコマンドの比較を表**12.1**に示します.

| | 機能 | サブコマンド | |
|---|---|---|---|
| | | SYMDEB | BEBUG |
| コンパチブルな機能 | ライン・アセンブル | A | A |
| | 指定領域の比較 | C | C |
| | データ・フィル | F | F |
| | 被テスト・プログラムの実行 | G | G |
| | 16進加減算の計算 | H | H |
| | ポート入力 | I | I |
| | ファイルのロード | L | L |
| | ブロック転送 | M | M |
| | ファイル名のセット | N | N |
| | ポート出力 | O | O |
| | デバッガを抜けてDOSに戻る | Q | Q |
| | レジスタの内容の表示／変更 | R | R |
| | 指定領域のサーチ | S | S |

| | | | |
|---|---|---|---|
| | トレース | T | T |
| | 逆アセンブル | U | U |
| | ファイルの書き込み | W | W |
| 拡張または追加された機能 | ブレーク・ポイントのセット | BP | |
| | ブレーク・ポイントの解除 | BC | |
| | ブレーク・ポイントの無効化 | BD | |
| | ブレーク・ポイントの有効化 | BE | |
| | ブレーク・ポイントのリスト | BL | |
| | ダンプ | D | D |
| | アスキー形式のダンプ | DA | |
| | バイト・ダンプ | DB | |
| | ワード・ダンプ | DW | |
| | ダブルワード・ダンプ | DD | |
| | 短い実数のダンプ | DS | |
| | 長い実数のダンプ | DL | |
| | 10バイト実数のダンプ | DT | |
| | データ・セット | E | E |
| | アスキー形式のデータ・セット | EA | |
| | バイト・データ・セット | EB | |
| | ワード・データ・セット | EW | |
| | ダブルワード・データ・セット | ED | |
| | 短い実数データ・セット | ES | |
| | 長い実数データ・セット | EL | |
| | 10バイト実数データ・セット | ET | |
| | スタック・フレームの内容表示 | K | |
| | 割り込みにも対応したトレース | P | |
| | 命令コードの表示形式の設定 | S | |
| | 指定したアドレスからソースラインを表示 | V | |
| | シンボル名とアドレスのリスト表示 | X | |
| | シンボル・マップまたはセグメントのセット | XO | |
| | シンボリック・アドレスに値をセット | Z | |
| | 値の表示とヘルプ機能 | ? | |
| | コメントの表示 | * | |
| | SYMDEBモードでMS-DOSコマンドの実行 | ! | |
| | 現在のソース・ラインの表示 | . | |
| | 入力のリダイレクト | < | |
| | 出力のリダイレクト | > | |
| | 入出力のリダイレクト | = | |

以下に拡張または追加されたサブコマンドの使い方を説明します．

● BP/BC/BD/BE/BL（Break Point set/Clear/Disable/Enable/List）

【書式】

BP[n]　アドレス　[回数]

BC　{リストまたは＊}

BD　{リストまたは＊}

BE　{リストまたは＊}

BL

Gコマンドで指定するブレーク・ポイントと違って，BPで指定したブレーク・ポイントはBCコマンドによりクリアしない限り，そのままです．

nは作成するブレーク・ポイント番号（0～9），＜リスト＞はブレーク・ポイント番号の列，＜回数＞は，何回目で実行を中止するかという，ブレーク・ポイントが採用されるまでに無視される回数です．

【例】

| | |
|---|---|
| －BP　CS：200 | …CS：200にブレーク・ポイントをセット |
| －BC　0␣3 | …ブレーク・ポイント0と3をクリア |
| －BD　1␣2␣3 | …ブレーク・ポイント1，2，3を一時的に無効にする |
| －BE　＊ | …すべてのブレーク・ポイントを有効化する |
| －BL | …ブレーク・ポイントの設定状況の表示 |

●D／DA／DB／DW／DD／DS／DL／DT

（Dump／Ascii／Bytes／Words／Double words／Short reals／Long reals／Ten byte reals）

【書式】

D／DA／DB／DW／DD／DS／DL／DT［レンジ］

2文字目に示す表示形態でダンプ出力します．Dのみ入力した場合は直前に行われた表示形態を引き継ぎます．一番最初の場合はバイト型が採用されます．Dのみの場合の書式には，レンジとの間には必ず空白が1つ必要となります．

【例】

－D　CS：100，11F　…表示形態は直前のものを引き継ぐ

－DB　CS：ABC，ABC＋16

…シンボルABC～ABC+16（バイト）をバイト型でダンプ

－DW　RESULT…シンボルRESULT（DS内）以後のデータをワード型でダンプ

● E／EA／EB／EW／ED／ES／EL／ET

（Enter／Ascii／Bytes／Words／Double words／Short reals／Long reals／Ten byte reals）

【書式】

E／EA／EB／EW／ED／ES／EL／ET アドレス［，リスト］

2文字目に示すデータ形態で指定アドレスにデータをセットします．Eのみ入力した場合はバイト型データとして扱われます．Eのみの場合の書式には，アドレスとの間には必ず空白が1つ必要となります．

【例】

－EB RESULT

…DS内のRESULT番地の内容を表示し，データ・セットの状態にする

● K（Stack Trace）

【書式】

K［値］

現在のスタック・フレームの内容を表示します．<値>を指定すると，その値の数まで引数が表示されます．

● P（PTrace）

【書式】

P［＝開始アドレス］［カウント］

Pコマンドはあらゆる呼び出しやソフトウェア割り込みからの復帰や実行を自動的に行うこと以外は，Tコマンドと同じです．

● S（Set Source Mode）

【書式】

$$S \begin{Bmatrix} - \\ \& \\ + \end{Bmatrix}$$

命令コードを表示する形式をセットします．＋はソースレベル，－は機械語レベル，＆はソースレベルと機械語レベルの併用モードです．

● V（View）

【書式】

V［アドレス］

高級言語用のサブコマンドで，ソース・テキストを行単位で表示する．

● X（eXamine symbol map）

【書式】

X[＊]

X?[マップ名!]［セグメント名:］［シンボル名］

現シンボル・マップ内のシンボル名とアドレスを表示します．＊を指定するとすべてのシンボルが表示されます．

● XO（Open Map）

【書式】

XO［マップ名！］セグメント名:

アクティブ・シンボル・マップとセグメント，またはそのどちらかをセットします．

● Z（Symbol Set）

【書式】

Z　シンボル　値

指定したシンボルのアドレスへ値をセットします．

●？（Display／Help）

**【書式】**

？［式］

？のみだとヘルプ機能が行われ，サブコマンド一覧が表示されます．式が指定されるとその値を表示します．

●＊（Comment）

**【書式】**

＊　コメント

コメントを出力装置にエコー・バックします．リダイレクション・コマンドと組み合わせて使用します．

●！（Shell Escape）

**【書式】**

！　MS-DOS コマンド

SYMDEB のモードから MS-DOS コマンドを実行します．

**【例】**

－!DIR　B：　…ドライブBのディレクトリを表示する

●（Source Line）

**【書式】**

高級言語のソース・テキストに対してのみ有効で，現在のソースラインを表示します．

●<／>／＝（Redirection）

【書式】

＜　入力装置

＞　出力装置

＝　入出力装置

I／Oリダイレクションを指定した装置名に向けます．AUX，CON，PRN などの装置名を用いることができます．

## 12.3 ／ MASM/LINK/LIBの追加事項

● MASM

アセンブル時に指定するスイッチに**表 12.2** に示す5つが追加されました．

●表12.2　ver 3.1 でMASMに追加されたスイッチ

| スイッチ | 機　　　能 |
|---|---|
| /ML | 名前に大文字，小文字の区別をつける |
| /MX | パブリック名と外部名に大文字，小文字の区別をつける |
| /R | 浮動小数点演算の数値演算プロセッサ用のコードの生成 |
| /E | 浮動小数点演算のエミュレート・コードの生成 |
| /A | セグメントをアルファベット順に出力 |

● LINK

リンク時に指定するスイッチに**表 12.3** に示す6つが追加されました．なお，ver 2.0ではスイッチ名として1文字～フル・スペルまでの入力を認めていましたが，ver 3.1では，先頭の2文字または3文字を指定しなければならないものもあります．

●表12.3 ver 3.1でLINKに追加されたスイッチ

| スイッチ | 省略形 | 機　　能 |
|---|---|---|
| /CPARMAXALLOC | /C | 最大割り当てスペースの設定 |
| /OVERLAYINTERRUPT | /O | オーバーレイ割り込みの設定 |
| /NOIGNORECASE | /NOI | 大文字，小文字の区別 |
| /NOGROUPASSOCIATION | /NOG | グループを取り除く |
| /SEGMENTS | /SE | 最大セグメントの設定 |
| /DOSSEG | /DO | DOSのセグメント配列の利用 |

● LIB

ライブラリ編集を行った結果を収める出力ファイル名を入力するようになりました．新しいLIBの起動例を以下に示します．

```
A>LIB

Microsoft Library Manager
Version 3.00 (C)Copyright Microsoft Corp 1983, 1984, 1985

Library name: MYLIB
Operations: +B:SMODUL1
List file:
Output library: MYLIB ←――――――出力ファイルを指定するようになった
```

LIBのコマンド・キャラクタとしては**表12.4**に示す2つが追加されました．

●表12.4 ver 3.1でLIBに追加されたコマンド・キャラクタ

| コマンド・キャラクタ | 機　　能 |
|---|---|
| －＋ | 置換．指定したオブジェクト・モジュールをライブラリから削除し，同名のオブジェクト・ファイル（.OBJ）をライブラリに入れます |
| －＊ | 移動．指定したオブジェクト・モジュールをライブラリから抜き取り，オブジェクト・ファイルに収めます |

# 12.4 プログラム・メインテナ(MAKE)

アセンブリ言語プログラムのモジュール別開発を行った結果，プログラムがいくつかのモジュールに分かれて，その中のあるモジュールに変更が生じたとき，すべてのモジュールについて再コンパイルするというのは無駄なことです．

そこで変更の必要のあるファイルを修正すると，どのファイルに影響があるかというモジュールの従属関係をメイク・ファイルというファイルにエディタで作成しておき，変更が生じたときにはメイク・ファイルを参照しながら自動的に必要なものだけをコンパイルまたはリンクすることができれば効率よく作業ができます．これを支援するユーティリティが**MAKE（プログラム・メインテナ）**です．

MAKEは一見バッチ・ジョブに似ています．ファイルの作成日を見ながら必要な処理を行う点では，より高度なバッチ・ジョブと考えても構いません．

以下にMAKEの使用例を示します．

**●メイク・ファイルの作成**

まず，メイク・ファイルの作り方を示します．メイク・ファイルは単一のジョブ（仕事）をいくつか連ねたもので，ジョブは，

```
目的ファイル名：　ソース・ファイル名リスト
        コマンド
```

の形を取り，各ジョブの間には1つの空白行を置きます．MAKEはこの目的ファイルとソース・ファイルの作成日を見て，もし目的ファイルの作成日がソース・ファイルの作成日より前であったなら，ソース側に変更があったものと判断し，コマンドを実行します．

メイク・ファイルの例を**図12.6**に示します．なお，メイク・ファイルには慣習上はファイルのタイプを付けません．

▼図12.6　メイク・ファイルの作成例

```
A>TYPE TESTM
TESTM.OBJ :    TESTM.ASM                     ┐
        MASM TESTM,TESTM;                    ┘ジョブ
                                     ←────────空白行
TESTS.OBJ :    TESTS.ASM
        MASM TESTS,TESTS;

TESTM.EXE :    TESTM.OBJ  TESTS.OBJ ←───ソース・ファイル名は空白で
        LINK TESTM+TESTS,TESTM;              区切る
```

今，TESTS .ASM を変更し，作成日が**図 12.7** のように，TESTS. OBJ より新しくなった場合に MAKE を実行するようすを**図 12.8** に示します．

▼図12.7　ファイル作成日

```
A>DIR TEST?.*

 ドライブ A のボリュームラベルはありません．
 ディレクトリは  A:¥

TESTM              157  85-01-23   2:36
TESTS    ASM       521  85-01-25   2:50 ←┐
TESTM    ASM       555  85-01-23   2:45   │ TESTS. ASM は
TESTM    OBJ       190  85-01-23   2:48   │ TESTS. OBJ より
TESTS    OBJ       151  85-01-23   2:48 ←┘ 新しい
TESTM    EXE       589  85-01-23   2:38
        6 個のファイルがあります．
    15360 バイトが使用可能です．
```

▼図12.8　MAKE の起動

```
A>MAKE TESTM ←──────────MAKEの起動     (↑ メイク・ファイル名)
MASM TESTS,TESTS;←──────────────TESTSのアセンブル
Microsoft MACRO Assembler  Version 3.00
(C)Copyright Microsoft Corp 1981, 1983, 1984


49698 Bytes free

Warning Severe
Errors  Errors
0       0
LINK TESTM+TESTS,TESTM;←────────リンク作業

Microsoft 8086 Object Linker
Version 3.00 (C) Copyright Microsoft Corp 1983, 1984, 1985
```

**図 12.8** に示すように MAKE は TESTM. ASM のアセンブルは行わずに，TESTS. ASM のアセンブルのみを行ってリンク作業に移ります．

# 巻　末

1．8086命令セット
2．MS-DOSファンクションコール一覧
3．MS-DOS ver 3.1における追加ファンクションコール

## ●8086命令セット（イラストラクションセット）

| ニーモニック | オペランド | 機　　　能 | フ | ラ | グ | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | O | D | I | T | S | Z | A | P | C |
| AAA | — | Ascii Adjust for Addition<br>加算結果のALの値をアンパック10進に補正する．上位ニブルは0 | U | | | | U | U | X | U | X |
| AAD | — | Ascii Adjust for Division<br>除算に先立ちALの値をアンパック10進に補正する． | U | | | | X | X | U | X | U |
| AAM | — | Ascii Adjust for Multiply<br>乗算結果のAXの値をアンパック10進に補正する． | U | | | | × | × | U | × | U |
| AAS | — | Ascii Adjust for Subtraction<br>減算結果のALの値をアンパック10進に補正する． | U | | | | U | U | × | U | × |
| ADC | reg, reg<br>reg, mem<br>mem, reg<br>reg, imm<br>mem, imm<br>acc, imm | ADd with Carry　キャリーを含めた加算<br>ADC　BX, XYZ…BX←BX＋(XYZ)＋CF<br>ADC　BX, 0005H…BX←BX＋0005H＋CF<br>ADC　[BX], 0005H…BXで示されるメモリに0005HとCFを加える． | × | | | | × | × | × | × | × |
| ADD | reg, reg<br>reg, mem<br>mem, reg<br>reg, imm<br>mem, imm<br>acc, imm | ADDition　キャリーを含めない加算<br>ADD　BX, XYZ…BX←BX＋(XYZ)<br>ADD　XYZ, 0005H…(XYZ)←(XYZ)＋0005H<br>ADD　[BX], DX…BXで示されるメモリにDXの内容を加える． | × | | | | × | × | × | × | × |
| AND | reg, reg<br>reg, mem<br>mem, reg<br>reg, imm<br>mem, imm<br>acc, imm | logical AND<br>AND　BX, CX…BXとCXのANDを取ってBXに入れる．<br>AND　XYZ, 00FFH…XYZと00FFHのANDを取ってXYZに入れる． | 0 | | | | × | × | U | × | 0 |
| CALL | near-proc<br>far-proc<br>memptr16<br>regptr16<br>memptr32 | CALL　プロシージャの呼び出し<br>CALL　ABC…プロシージャABCをコール<br>CALL　AX…AXで示される番地をコール<br>CALL　SUB [SI]…SUB [SI]によってアドレッシングされるメモリの内容が示す番地をコールする． | | | | | | | | | |
| CBW | — | Convert Byte to Word<br>ALのバイト・データを符号ビットを含めてAXのワード・データに拡張<br>AL＜80HならばAH←0<br>AL≧80HならばAH←FFH | | | | | | | | | |
| CLC | — | CLear Carry flag<br>キャリー・フラグをクリアする | | | | | | | | | 0 |
| CLD | — | CLear Direction flag | | 0 | | | | | | | |
| CLI | — | CLear Interrupt flag | | | 0 | | | | | | |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| CMC | — | CoMplement Carry flag<br>キャリー・フラグを反転する. | | | | | | | | | × |
| CMP | reg, reg<br>reg, mem<br>mem, reg<br>reg, imm<br>mem, imm<br>acc, imm | CoMpare　2つのオペランドを比較する<br>CMP　BL, 5…BLと5を比較し，結果によりフラグをセット<br>CMP　XYZ, AX…XYZの内容とAXを比較 | × | | | | × | × | × | × | × |
| CMPSB<br>CMPSW | —<br>— | CoMPare String Byte/Word<br>(DS：SI)と(ES：DI)の内容を比較し，その結果でフラグがセットされる．Dフラグに応じてSI，DIは増減する | × | | | | × | × | × | × | × |
| CWD | — | Convert Word to Double word<br>AXのワード・データを符号を含めてDX：AXのダブルワード・データに変換<br>AX＜8000HならばDX←0<br>AX≧8000HならばDX←FFFFH | | | | | | | | | |
| DAA | — | Decimal Adjust for Addition<br>加算結果のALの値をパック10進に補正 | U | | | | × | × | × | × | × |
| DAS | — | Decimal Adjust for Subtraction<br>減算結果のALの値をパック10進に補正 | U | | | | × | × | × | × | × |
| DEC | reg<br>mem | DECrement　オペランドの内容が－1される<br>DEC　BX…BX←BX－1 | × | | | | × | × | × | × | |
| DIV | reg<br>mem | DIvision<br>AXまたはDX：AXの内容をオペランドの内容で除算する．商はALまたはAX，余りはAHまたはDXに格納する | U | | | | U | U | U | U | U |
| ESC | imm, mem<br>imm, reg | ESCape　エスケープ命令 | | | | | | | | | |
| HLT | — | HALT　停止 | | | | | | | | | |
| IDIV | reg<br>mem | Integer DIVision<br>符号付きの除算であることを除けばDIVと同じ | U | | | | U | U | U | U | U |
| IMUL | reg<br>mem | Integer MULtiply<br>符号付きの乗算であることを除けばMULと同じ | × | | | | U | U | U | U | × |
| IN | acc, imm8<br>acc, DX | INput<br>オペランドで示されるポートからALまたはAXにデータを入力<br>IN　AL, 0E0H…E0HポートのデータをALに入力 | | | | | | | | | |
| INC | reg<br>mem | INCrement　オペランドの内容を＋1する<br>INC　XYZ…XYZの内容を＋1 | × | | | | × | × | × | × | |
| INT | imm8 | INTerrupt<br>割り込み．INT 3(ブレーク・ポイント用)だけは1バイト命令 | | | 0 | 0 | | | | | |
| INTO | — | INTerrupt if Overflow | | | 0 | 0 | | | | | |
| IRET | — | Interrupt RETurn | R | R | R | R | R | R | R | R | R |
| JA/<br>JNBE | Short-label | Jump if Above/<br>Jump if Not Below nor Equal | | | | | | | | | |
| JAE/<br>JNB | Short-label | Jump if Adove or Equal/<br>Jump if Not Below | | | | | | | | | |
| JB/ | Short-label | Jump if Below/ | | | | | | | | | |

| | | |
|---|---|---|
| JNAE | | Jump if Not Above nor Equal |
| JBE/<br>JNA | Short-label | Jump if Below or Equal/<br>Jump if Not Above |
| JC | Short-label | Jump if Carry |
| JCXZ | Short-label | Jump if CX is Zero<br>レジスタCXが0ならばジャンプする |
| JE/<br>JZ | Short-label | Jump if Equal/<br>Jump if Zero |
| JG/<br>JNLE | Short-label | Jump if Greater/<br>Jump if Not Less nor Equal |
| JGE/<br>JNL | Short-label | Jump if Greater or Equal/<br>Jump if Not Less |
| JL/<br>JNGE | Short-label | Jump if Less/<br>Jump if Not Greater nor Equal |
| JLE/<br>JNG | Short-label | Jump if Less or Equal/<br>Jump if Not Greater |
| JMP | Short-label<br>near-label<br>far-label<br>memptr16<br>regptr16<br>memptr32 | JuMP　無条件ジャンプ<br>JMP　ABC…ABCへジャンプする<br>JMP　AX…AXが示す番地へジャンプする<br>JMP　ABC [SI]…ABC [SI]によってアドレッシングされるメモリの内容が示す番地へジャンプする |
| JNC | Short-label | Jump if Not Carry |
| JNE/<br>JNZ | Short-label | Jump if Not Equal/<br>Jump if Not Zero |
| JNO | Short-label | Jump if Not Overflow |
| JNP/<br>JPO | Short-label | Jump if Not Parity/<br>Jump if Parity Odd |
| JNS | Short-label | Jump if Not Sign |
| JO | Short-label | Jump if Overflow |
| JP/<br>JPE | Short-label | Jump if Parity/<br>Jump if Parity Even |
| JS | Short-label | Jump if Sign |
| LAHF | — | Load AH from Flags<br>フラグの内容をAHの特定ビットへロードする |
| LDS | reg16<br>mem32 | Load Data Segment register<br>reg16←(mem32)，DS←(mem32＋2) |
| LEA | reg16<br>mem16 | Load Effective Address<br>mem16のオフセット値をreg16へロードする |
| LES | reg16<br>mem32 | Load Extra-Segment register<br>reg16←(mem32)，ES←(mem32＋2) |
| LOCK | — | LOCK bus<br>プリフィックス命令 |
| LODSB<br>LODSW | — | LOaD String Byte/Word<br>(SI)の内容をALまたはAXにロードする．Dフラグに応じてバイト型なら±1，ワード型なら±2の範囲でSIは増減する． |
| LOOP | Short-label | LOOP CX times |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | CXを－1しながらCXが0になるまで指定したラベルへジャンプする | | | | | | | | | |
| LOOPE/<br>LOOPZ | Short-label | LOOP if Equal/LOOP if Zero<br>ゼロ・フラグが1，かつCX≠0ならば指定したラベルへジャンプする | | | | | | | | | |
| LOOPNE/<br>LOOPNZ | Short-label | LOOP if Not Equal/LOOP if Not Zero<br>ゼロ・フラグが0，かつCX≠0ならば指定したラベルへジャンプする | | | | | | | | | |
| MOV | reg, reg<br>mem, reg<br>reg, mem<br>mem, imm<br>reg, imm<br>acc, mem<br>mem, acc<br>sreg, reg16<br>sreg, mem16<br>reg16, sreg<br>mem, sreg | MOVe　転送命令<br>MOV　AX, CX…AX←CX<br>MOV　AX, [BX]…BXで示されるメモリの内容をAXにロードする<br>MOV　BX, 0123H…BXに0123Hという値をロードする<br>MOV　BX, XYZ…BXにXYZ番地の内容をロードする<br>MOV　DS, AX…DSレジスタにAXの値をロードする | | | | | | | | | |
| MOVSB<br>MOVSW | — | MOVe String Byte/Word<br>DS：SIで示されるメモリのデータをES：DIで示されるメモリにロードする．Dフラグに応じてバイト型なら±1，ワード型なら±2の範囲でSI, DIは増減する． | | | | | | | | | |
| MUL | reg<br>mem | MULtiply accumulator by register or memory<br>ALまたはAXとオペランドを乗算し，結果をAXまたはDX：AXに格納する | × | | | | U | U | U | U | × |
| NEG | reg<br>mem | NEGate<br>オペランドの2の補数，つまり符号反転を行う． | × | | | | × | × | × | × | × |
| NOP | — | No OPeration | | | | | | | | | |
| NOT | reg<br>mem | logical NOT<br>オペランドの1の補数，つまりビット反転を行う． | | | | | | | | | |
| OR | reg, reg<br>reg, mem<br>mem, reg<br>reg, imm<br>mem, imm<br>acc, imm | logical OR<br>OR　BX, CX…BXとCXのORを取ってBXに入れる<br>OR　XYZ, 00FFH…XYZの内容と00FFHのORを取ってXYZに入れる | 0 | | | | × | × | U | × | 0 |
| OUT | imm8, acc<br>DX, acc | OUTput<br>オペランドで示されるポートへALまたはAXのデータを送る<br>OUT　DX, AL…ALの1バイトデータをDXで示されるポートへ出力 | | | | | | | | | |
| POP | reg16<br>sreg<br>mem16 | POP words off stack<br>スタックから値を取り出す．sregの中でCSだけは指定できない．<br>POP　XYZ…スタックからデータを取り出し，XYZに入れる | | | | | | | | | |
| POPF | — | POP Flags off stack | R | R | R | R | R | R | R | R | R |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | スタックからデータを取り出してFレジスタへ入れる | | | | | | | | | |
| PUSH | reg16<br>sreg<br>mem16 | PUSH words onto stack<br>オペランドの内容をスタックに積む<br>PUSH CS…CSの内容をスタックに積む | | | | | | | | | |
| PUSHF | — | PUSH Flags onto stack<br>Fレジスタの内容をスタックに積む | | | | | | | | | |
| RCL | reg, 1<br>reg, CL<br>mem, 1<br>mem, CL | Rotate Left through Carry<br>キャリーを含めて左へオペランドで示された回数だけローテイトする<br>RCL XYZ, CL…XYZの内容をCLで示される回数，左へローテイトする | ×<br>U<br>×<br>U | | | | | | | | ×<br>×<br>×<br>× |
| RCR | reg, 1<br>reg, CL<br>mem, 1<br>mem, CL | Rotate Right through Carry<br>キャリーを含めて右へオペランドで示された回数だけローテイトする<br>RCR XYZ, CL…XYZの内容をCLで示される回数，右へローテートする | ×<br>U<br>×<br>U | | | | | | | | ×<br>×<br>×<br>× |
| REP | — | REPeat string operation<br>REPの後にくるストリング命令をCXが0になるまで繰り返す<br>REP MOVSB | | | | | | | | | |
| REPE/<br>REPZ | — | REPeat string operation while Equal/<br>REPeat string operation while Zero | | | | | | | | | |
| REPNE/<br>REPNZ | — | REPeat string operation while Not Equal/<br>REPeat string operation while Not Zero | | | | | | | | | |
| RET | —<br>imm | RETurn<br>CALLでスタックにPUSHされた復帰番地に戻る．もし，immが指定されていれば，その値がSPに加えられる | | | | | | | | | |
| ROL | reg, 1<br>reg, CL<br>mem, 1<br>mem, CL | Rotate Left<br>左へオペランドで示された回数だけローテイトする．キャリーは最位ビットで置き換えられる<br>ROL AL, 1…ALの内容を1ビット左へローテイトする | ×<br>U<br>×<br>U | | | | | | | | ×<br>×<br>×<br>× |
| ROR | reg, 1<br>reg, CL<br>mem, 1<br>mem, CL | Rotate Right<br>右へオペランドで示された回数だけローテイトする．キャリーは最下位ビットで置き換えられる<br>ROR XYZ, 1…XYZの内容を1ビット右へローテイトする | ×<br>U<br>×<br>U | | | | | | | | ×<br>×<br>×<br>× |
| SAHF | — | Store AH into Flag<br>AHの特定ビットがFレジスタに転送される | | | | | R | R | R | R | R |
| SAL/<br>SHL | reg, 1<br>reg, CL<br>mem, 1<br>mem, CL | Shift Arithmetic Left/SHift logical Left<br>左へオペランドで示された回数だけシフトする．キャリーは最上位ビットで置き換えられる．最下位ビットは0になる．<br>SHL AX, CL…AXの内容がCLビット左へシフト | × | | | | × | × | U | × | × |
| SAR | reg, 1<br>reg, CL<br>mem, 1<br>mem, CL | Shift Arithmetic Right<br>右へオペランドで示された回数だけシフトする．キャリーは最下位ビットで置き換えられる．最上位ビットはそのまま．<br>SAR AL, 1…ALの内容が1ビット右へシフト | × | | | | × | × | U | × | × |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SBB | reg, reg<br>reg, mem<br>mem, reg<br>reg, imm<br>mem, imm<br>acc, imm | SuBtract with Borrow<br>キャリーを含めた減算<br>SBB　AX, BX…AX←AX−BX−CF | × | | | | × | × | × | × | × |
| SCASB<br>SCASW | —<br>— | SCAn String Byte/Word<br>ALまたはAXとES：DIで示されるメモリ内容が比較され，その結果でフラグがセットされる．Dフラグに応じてDIは，バイト型なら±1，ワード型なら±2の範囲で増減する． | × | | | | × | × | × | × | × |
| SHR | reg, 1<br>reg, CL<br>mem, 1<br>mem, CL | SHift logical Right<br>右へオペランドで示された回数だけシフトする．キャリーは最下位ビットで置き換えられる．最下位ビットは0になる． | × | | | | × | × | U | × | × |
| STC | — | SeT Carry flag | | | | | | | | | 1 |
| STD | — | SeT Direction flag | | 1 | | | | | | | |
| STI | — | SeT Interrupt enable flag | | | 1 | | | | | | |
| STOSB<br>STOSW | —<br>— | STOre String Byte/Word<br>ALまたはAXの内容をES：DIで示されるメモリにロードする．Dフラグに応じてDIは，バイト型なら±1，ワード型なら±2の範囲で増減する． | | | | | | | | | |
| SUB | reg, reg<br>reg, mem<br>mem, reg<br>reg, imm<br>mem, imm<br>acc, imm | SUBtract<br>キャリーなしの減算<br>SUB　XYZ, BX…(XYZ)←(XYZ)−BX<br>SUB　XYZ, 00FFH…(XYZ)←(XYZ)−00FFH | × | | | | × | × | × | × | × |
| TEST | reg, reg<br>mem, reg<br>reg, imm<br>mem, imm<br>acc, imm | TEST<br>第1オペランドと第2オペランドのANDを取り，その結果でフラグが変化し，第1オペランドの内容は変化しない．<br>TEST　AX, BX…AXとBXのANDを取る | 0 | | | | × | × | U | × | 0 |
| WAIT | — | WAIT<br>プロセッサを一時停止の状態にする | | | | | | | | | |
| XCHG | reg, reg<br>mem, reg<br>AX, reg16 | eXCHanGe<br>第1オペランドと第2オペランドの内容を交換する<br>XCHG　XYZ, BX…XYZの内容とBXを交換 | | | | | | | | | |
| XLAT | — | Translate<br>BX：ALで示されるテーブルの内容をALにロードする | | | | | | | | | |
| XOR | reg, reg<br>reg, mem<br>mem, reg<br>reg, imm<br>mem, imm<br>acc, imm | logical eXclusive OR<br>XOR　BX, 0FFFFH…BXと0FFFFHとのXORが取られて結果がBXに入る | 0 | | | | × | × | U | × | 0 |

## ●表中のオペランド記号

reg　…汎用レジスタ　AL, AH, BL, BH, CL, CH, DL, DH, AX, BX, CX, DX, SP, BP, SI, DI
reg 8　…8ビット長レジスタ　AL, AH, BL, BH, CL, CH, DL, DH
reg16　…16ビット長レジスタ　AX, BX, CX, DX, SP, BP, SI, DI,
sreg　…セグメント・レジスタ　CS, DS, ES, SS
mem　…8/16ビット長メモリ
mem16…16ビット長メモリ
mem32…32ビット長メモリ
imm16…8/16ビット長即値
imm 8　…8ビット長即値
imm　…16ビット長即値
imm32…32ビット長即値
acc　…アキュムレータ　AL, AX
(ニーモニックのレベルでは, accはregの中に含めて考えてもよいが, accを用いた場合の方が他のレジスタを用いた場合よりも短いバイト数のコードを得ることができる)
near-proc　…ニア・プロシージャ
far-proc　ファー・プロシージャ
regpr16　レジスタ・ポインタ(16)
memptr16　メモリ・ポインタ(16)
memptr32　メモリ・ポインタ(32)
Short-label…ショート・ラベル

## ●メモリ・アドレッシング

[BX] + [SI]
[BX] + [DI]
[BP] + [SI]
[BP] + [DI]
[ SI ]
[ DI ]
disp16
[BX]
[BX] + [SI] +disp 8
[BX] + [DI] +disp 8
[BP] + [SI] +disp 8
[BP] + [DI] +disp 8
[ SI ] + disp 8
[ DI ] + disp 8
[BP] + disp 8
[BX] + disp 8

[BX] + [SI] + disp16
[BX] + [DI] + disp16
[BP] + [SI] + disp16
[BP] + [DI] + disp16
[ SI ] + disp16
[ DI ] + disp16
[BP] + disp16
[BX] + disp16

注) disp 8…8ビット長ディスプレイスメント
　　disp16…16ビット長ディスプレイスメント

## ●フラグの記号

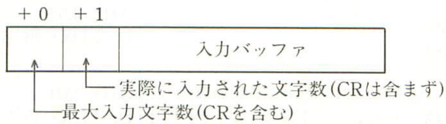
O …オーバーフロー・フラグ
D …ディレクション・フラグ
I …割り込みイネーブル・フラグ
T …トラップ・フラグ
S …サイン・フラグ
Z …ゼロ・フラグ
A …補助キャリー・フラグ
P …パリティ・フラグ
C …キャリーフラグ

## ●フラグの動作

空白…変化なし
0 …0にリセット
1 …1にセット
X …結果に従って, セット/リセットされる
U …不定
R …それ以前に退避させた値がリストアされる

## ■巻末付録2■

### ●MS-DOSファンクションコール(システムコール)一覧

| Hex | 機　能 | 入力パラメータ | リターン情報 |
|---|---|---|---|
| 0 | プログラムの終了 | AH←00H<br>CS←PSPのセグメントアドレス | なし |
| 1 | キーボードから1文字入力<br>内部ループ型でエコーバックあり | AH←01H | AL←入力された文字 |
| 2 | スクリーンへの1文字出力 | AH←02H<br>DL←スクリーン出力する文字 | なし |
| 3 | 補助入力装置から1文字入力 | AH←03H | AL←入力された文字 |
| 4 | 補助出力装置への1文字出力 | AH←04H<br>DL←出力すべき文字 | なし |
| 5 | プリンタへの1文字出力 | AH←05H<br>DL←出力すべき文字 | なし |
| 6 | DLにセットする値により異る：キーボードからの入力．内部ループせずエコーバックなし CTRL+C を受け付けない | AH←06H<br>DL←0FFH | AL←キーボードから入力された文字<br>(入力がなければ0(ヌル)が入る) |
| 6 | DLにセットする値により異る：スクリーンへの出力 | AH←06H (FFH以外)<br>DL←スクリーンに出力する文字 | なし |
| 7 | 直接コンソール入力<br>内部ループ型でエコーバックなし<br>CTRL+Cを受け付けない | AH←07H | AL←入力された文字 |
| 8 | キーボード入力<br>内部ループ型でエコーバックなし． | AH←08H | AL←入力された文字 |
| 9 | 文字列のスクリーンへの出力．<br>文字列の終りは"$"とする． | AH←09H<br>DS：DX←文字列のオフセット | なし |
| 0A<br>(10) | バッファに文字列を入力．<br>入力終りは↵．↵入力前なら入力文字の修正が行える． | AH←0AH<br>DS：DX←入力バッファのオフセット<br> | なし |
| 0B<br>(11) | キーボードステータスの検査 | AH←0BH | AL←FFH：タイプヘッドバッファに文字が入っている<br>0　：入っていない |
| 0C<br>(12) | バッファを空にした後ALに与えたファンクション番号のキーボード入力を行う．規定外の値がALにセットされれば入力は行われない． | AH←0CH<br>AL←1, 6, 7, 8, 0AH | AL←入力された文字<br>(規定外の値をALにセットしたときはAL←0となる) |
| 0D<br>(13) | ディスク・リセット | AH←0DH | なし |
| 0E<br>(14) | カレント・ディスクの指定 | AH←0EH　DL←ドライブ番号<br>(0=A, 1=B…) | AL←指定されたドライブ番号 |
| 0F<br>(15) | ファイルのオープン | AH←0FH<br>DS：DX←FCBのオフセット | AL←0：ディレクトリエントリが存在する<br>FFH：ディレクトリエントが存在しない |
| 10<br>(16) | ファイルのクローズ | AH←10H<br>DS：DX←FCBのオフセット | AL←0：ディレクトリ・エントリが在存する<br>FFH　：ディレクトリ・エントリが存在しない |

| | | | |
|---|---|---|---|
| 11<br>(17) | 最初のディレクトリ・エントリを検索 | AH←11H<br>DS：DX←FCBのオフセット | AL←0：ディレクトリ・エントリが存在する<br>FFH：ディレクトリ・エントリが存在しない |
| 12<br>(18) | 次のディレクトリ・エントリを検索 | AH←12H<br>DS：DX←FCBのオフセット | 同上 |
| 13<br>(19) | ファイルの削除 | AH←13H<br>DS：DX←FCBのオフセット | 同上 |
| 14<br>(20) | シーケンシャル・リード | AH←14H<br>DS：DX←FCBのオフセット | AL←0：正常<br>1：EOF<br>2：DTAが不足<br>3：最後のレコード |
| 15<br>(21) | シーケンシャル・ライト | AH←15H<br>DS：DX←FCBのオフセット | AL←0：正常<br>1：ディスクフル<br>2：DTAが不足 |
| 16<br>(22) | ファイルの生成 | AH←16H<br>DS：DX←FCBのオフセット | AL←0：空のディレクトリが存在する<br>FFH：空のディレクトリが存在しない |
| 17<br>(23) | ファイル名の変更 | AH←17H<br>DS：DX←FCBのオフセット<br>(FCBの先頭に旧いファイル名<br>オフセット11Hから新しいファイル名) | AL←0：ディレクトリ・エントリが存在する<br>FFH：ディレクトリ・エントリが存在しない |
| 19<br>(25) | カレント・ディスク番号を得る | AH←19H | AL←カレント・ディスク<br>(0＝A, 1＝B…) |
| 1A<br>(26) | ディスク転送アドレスのセット | AH←1AH<br>DS：DX←DTAアドレス | なし |
| 21<br>(33) | ランダム・リード | AH←21H<br>DS：DX←FCBのオフセット | AL←0：正常<br>1：EOF<br>2：DTAが不足<br>3：最後のレコード |
| 22<br>(34) | ランダム・ライト | AH←22H<br>DS：DX←FCBのオフセット | AL←0：正常<br>1：ディスク・フル<br>2：DTAが不足 |
| 23<br>(35) | ファイル・レコード数の取得．相対レコードフィールド(オフセット21H)にファイルのレコード数が得られる． | AH←23H<br>DS：DX←FCBのオフセット<br>レコードサイズフィールド(オフセット0EH)<br>←レコードサイズ(ディフォルト128) | AL←0：ディレクトリ・エントリが存在する<br>FFH：ディレクトリ・エントリが存在しない |
| 24<br>(36) | 相対レコードフィールド(オフセット21H)がカレント・ブロック(オフセット0CH)とカレント・レコード(オフセット20H)と同じファイル・アドレスにセットされる． | AH←24H<br>DS：DX←FCBのオフセット | なし |
| 25<br>(37) | 割り込みベクトルのセット | AH←25H<br>AL←割り込みタイプ番号<br>DS：DX←割り込みルーチンのオフセット | なし |
| 27<br>(39) | ランダム・ブロック・リード | AH←27H<br>DS：DX←FCBのオフセット<br>CX←リードするレコード数 | AL←0：正常<br>1：EOF<br>2：セグメントの終り<br>3：最後のレコード<br>CX←リードされたレコード数 |

| | | | |
|---|---|---|---|
| 28<br>(40) | ランダム・ブロック・ライト | AH←28H<br>DS：DX←FCBのオフセット<br>CX←ライトするレコード数 | AL←0：正常<br>1：ディスクフル<br>2：セグメントの終り<br>CX←ライトされたレコード数 |
| 29<br>(41) | ファイル名の解析 | AH←29H<br>AL←解析の制御(注1)<br>DS：SI←解析すべきストリング<br>ES：DI←FCBのオフセット | AL←0：ワイルド・カード使用されている<br>1：ワイルド・カード使用されていない<br>FFH：ドライブ名が無効<br>DS：SI←解析されたストリングの次にくる最初のバイト<br>ES：DI←FCBのオフセット |
| 2A<br>(42) | 日付を得る | AH←2AH | CX←年(1980～2079)<br>DH←月(1～12)<br>DL←日(1～31)<br>AL←曜日(0：日，……6：土) |
| 2B<br>(43) | 日付をセット | AH←2BH<br>CX←年(1980～2079)<br>DH←月(1～12)<br>DL←日(1～31) | AL←0：日付はセットされた<br>FFH：日付にエラーあった |
| 2C<br>(44) | 時刻を得る | AH←2CH | CH←時(0～23)<br>CL←分(0～59)<br>DH←秒(0～59)<br>DL←1/100移(0～99) |
| 2D<br>(45) | 時刻をセット | AH←2DH<br>CH←時(0～23)<br>CL←分(0～59)<br>DH←秒(0～59)<br>DL←1/100秒(0～99) | AL←0：時刻はセットされた<br>FFH：時刻は無効 |
| 2E<br>(46) | ベリファイフラグのセット/リセット | AH←2EH<br>AL←0(ベリファイを行わない，ディフォルト)<br>1(ベリファイを行う)<br>DL←0 | なし |
| 2F<br>(47) | ディスク転送アドレスを得る | AH←2F | ES：BX←ディスク転送アドレス |
| 30<br>(48) | DOSバージョンを得る | AH←30H | AL←バージョン番号の整数部<br>AH←バージョン番号の小数部 |
| 31<br>(49) | キープ・プロセス<br>(プログラムの常駐終了) | AH←31H<br>AL←抜け出しコード<br>DX←パラグラフでのメモリサイズ | なし |
| 33<br>(51) | CTRL-C検査のセット/リセット | AH←33H<br>AL←0：現ステートの問合せ<br>1：ステートのセット<br>DL←0：オフ<br>1：オン (ステートセットの場合) | DL←0：オフ<br>1：オン (ステートの問合せの場合) |
| 35<br>(53) | 割り込みベクトルを得る | AH←35H<br>AL←割り込み番号 | ES：BX←割り込みルーチンのオフセット |
| 36<br>(54) | ディスクのフリースペースを得る | AH←36H<br>DL←ドライブ番号(0：カレント，1：A…) | BX←使用可能なクラスタ数<br>DX←1ドライブ当りのクラスタ数 |

| | | | |
|---|---|---|---|
| | | | CX←1セクタ当りのバイト数<br>AX←1クラスタ当りのセクタ数(FFFFが返されればドライブが無効であった事を示す. |
| 38<br>(56) | 国別情報を得る | AH←38H<br>DS:DX←バッファ(32バイト)オフセット<br>AL←0:現在の国<br>0:USA規格<br>51H:日本規格 | キャリーがセットされない場合<br>→バッファにデータが得られた<br>キャリーがセットされた場合<br>AX←2:データがない |
| 39<br>(57) | サブディレクトリの作成 | AH←39H<br>DS:DX←パス名の位置 | キャリーがセットされない場合<br>→正常<br>キァリーがセットされた場合<br>AX←3:パスは存在しない<br>5:アクセスは否定された |
| 3A<br>(58) | ディレクトリ・エントリの削除 | AH←3AH<br>DS:DX←パス名の位置 | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←3:パスは存在しない<br>5:アクセスは存在された<br>16:現在のディレクトリであった. |
| 3B<br>(59) | カレント・ディレクトリの変更 | AH←3BH<br>DS:DX←パス名の位置 | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←3:パスは存在しない |
| 3C<br>(60) | ファイルの生成 | AH←3CH<br>DS:DX←パス名の位置<br>CX←ファイルの属性(注4) | キャリーがセットされない場合<br>AX←ファイル・ハンドル<br>キャリーがセットされた場合<br>AX←3:パスは存在しない<br>4:オープされているファイルが多すぎる<br>5:アクセスは否定された |
| 3D<br>(61) | ファイルのオープン | AH←3DH<br>DS:DX←パス名の位置<br>AL←0:リードオープン<br>1:ライトオープン<br>2:リード/ライトオープン | キャリーがセットされない場合<br>AX←ファイル・ハンドル<br>キャリーがセットされた場合<br>AX←2:ファイルが存在しない<br>4:オープンされているファイルが多すぎる<br>5:アクセスは否定された<br>12:無効なアクセス |
| 3E<br>(62) | ファイル・ハンドルのクローズ | AH←3EH<br>BX←ファイル・ハンドル | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←6:無効なハンドル |
| 3F<br>(63) | ファイル/デバイス・リード | AH←3FH<br>DS:DX←バッファの位置<br>CX←リードするバイト数<br>BX←ファイル・ハンドル | キャリーがセットされない場合<br>AX←リードされたバイト数<br>キャリーがセットされた場合<br>AX←5:エラーがセットされる<br>6:ハンドルが無効 |
| 40<br>(64) | ファイル/デバイス・ライト | AH←40H<br>DS:DX←バッファの位置<br>CX←ライトするバイト数 | キャリーがセットされない場合<br>AX←ライトされたバイト数<br>キャリーがセットされた場合 |

| | | | |
|---|---|---|---|
| | | BX←ファイル・ハンドル | AX←5：アクセスは否定された<br>6：無効なハンドル |
| 41<br>(65) | ディレクトリ・エントリの削除 | AH←41H<br>DS：DX←パス名の位置 | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←2：ファイルが存在しない<br>5：アクセスは否定された |
| 42<br>(66) | ファイル・ポインタの移動 | AH←42H<br>CX：DX←移動するバイト数<br>BX←ファイル・ハンドル<br>AL←0：ファイルが先頭からオフセット位置へ<br>1：現ロケーション＋オフセット位置へ<br>2：ファイルの終り＋オフセット位置へ | キャリーがセットされない場合<br>DX：AX←新規のポインタ・ロケーション<br>キャリーがセットされた場合<br>AX←1：無効なファンクション<br>6：無効な処理 |
| 43<br>(67) | アトリビュート(属性)の変更 | AH←43H<br>DS：DX←パス名の位置<br>CX←セットすべき属性(AL＝1のとき)(注4)<br>AL←0：ファイルの現在の属性を返す<br>1：CXで指定した属性にセット | キャリーがセットされない場合<br>CX←属性(AL＝0のとき)<br>キャリーがセットされた場合<br>AX←1：無効なファンクション<br>3：パスが存在しない<br>5：アクセスは否定された |
| 44<br>(68) | デバイスに対するI/Oコントロール | AH←44H<br>BX←ファイル・ハンドル<br>BL←ドライブ(AL＝4,5のコールの場合. 0＝カレント，1＝A…)<br>DS：DX←データまたはバッファのアドレス<br>CX←リード/ライトすべきバイト数<br>AL←I/Oコントロールファンクション(注2) | キャリーがセットされない場合<br>AL＝2，3，4，5の場合<br>AX←転送されたカウント<br>AL＝6，7の場合<br>AL←0：ノットレディ<br>FFH：レディ<br>キャリーがセットされた場合<br>AX←1：無効なファンクション<br>5：アクセスは否定された<br>6：無効なハンドル<br>13：無効なデータ |
| 45<br>(69) | ファイル・ハンドルの二重化<br>すでにオープンされているファイル・ハンドルを取り出し，同じファイルを扱う新規のハンドルを返す | AH←45H<br>BX←ファイル・ハンドル | キャリーがセットされない場合<br>AX←新規のファイル・ハンドル<br>キャリーがセットされた場合<br>AX←4：オープンされているファイルが多すぎる<br>6：無効なハンドル |
| 46<br>(70) | ハンドラの二重化<br>すでにオープンされているファイル・ハンドルを取り出し，同じ位置の同じファイルを示す新規のハンドルを返す | AH←46H<br>BX←既存のファイル・ハンドル<br>CX←新規のファイル・ハンドル | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←4：オープンされているファイルが多すぎる<br>6：無効な処理 |
| 47<br>(71) | 指定したドライブのカレント・ディレクトリを返す. | AH←47H<br>DS：SI←64バイトのバッファへのポインタ<br>DL←ドライブ番号 | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←15：無効なドライブ |
| 48<br>(72) | メモリの割り当て | AH←48H<br>BX←割り当てるメモリの大きさ(パラグラフ) | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←7：メモリ中のデータが破壊されている<br>8：十分な大きさのメモリがない |

| | | | |
|---|---|---|---|
| 49<br>(73) | 割り当てられたメモリの解放 | AH←49H<br>ES←解放すべき領域のセグメント値(パラグラフ) | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←7:メモリ中のデータが破壊されている<br>9:無効なブロック |
| 4A<br>(74) | 割り当てられたメモリブロックの大きさを拡大/縮小する | AH←4AH<br>ES←メモリブロックのセグメント値<br>BX←メモリブロックの先頭アドレス(パラグラフ) | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←7:メモリ中のデータが破壊されている<br>8:十分な大きさのメモリがない<br>9:無効なブロック |
| 4B<br>(75) | あるプログラムで他のプログラムをロード/実行する | AH←4BH<br>DS:DX←パス名の位置<br>ES:BX←ロードのためのパラメータブロック<br>AL←0:ロードと実行<br>3:ロードのみ | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←1:無効なファンクション<br>2:ファイルが存在しない<br>8:十分な大きさのメモリがない<br>10:不正な環境<br>11:不正なフォーマット |
| 4C<br>(76) | 現在のプロセスを終了し，コールを行ったプロセスに戻る | AH←4CH<br>AL←リターンコード | なし |
| 4D<br>(77) | 子プロセスで指定した抜け出しコードを1回だけ返す | AH←4DH | AX←抜け出しコード |
| 4E<br>(78) | 一致するファイル名の検索 | AH←4EH<br>DS:DX←パス名の位置<br>CX←ファイルの属性(注4) | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←2:ファイルが存在しない<br>18:これ以上ファイルが存在しない |
| 4F<br>(79) | 次に一致するファイル名の検索 | AH←4FH | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←18:これ以上ファイルが存在しない |
| 54<br>(84) | ベリファイ・フラグの現在値を返す. | AH←54H | AL←ベリファイ・フラグの値<br>(0=OFF, 1=ON) |
| 56<br>(86) | ディレクトリ・エントリの移動(他のパスにファイル名を変更する.) | AH←56H<br>DS:DX←既存のパス名の位置<br>ES:DI ←新規のパス名の位置 | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←2:ファイルが存在しない<br>5:アクセスは否定された<br>17:同じ装置でない |
| 57<br>(87) | ファイルの日付/時刻を得る/セットする. | AH←57H<br>AL←0:得る<br>1:セットする<br>BX←ファイル・ハンドル<br>CX←セットすべき時刻(AL=1のとき)<br>DX←セットすべき日付(AL=1のとき) | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←1:無効なファンクション<br>6:無効なハンドル |

●ファイル名解析(ファンクション29Hの制御ビット)(注1)

| ビット | 値 | 意味 |
|---|---|---|
| 0 | 0 | ファイル分離記号を検出した場合，すべての解析が停止する． |
| | 1 | 先行する分離記号は無視される． |
| 1 | 0 | ストリングにドライブ番号が入っていない場合，FCB内のドライブ番号はカレント・ドライブにセット |
| | 1 | ストリングにドライブ番号が入っていない場合，FCB内のドライブ番号は変更されない． |
| 2 | 0 | ストリングにファイル名が入っていない場合，FCB内のファイル名は8つのスペースにセット |
| | 1 | ストリングにファイル名が入っていない場合，FCB内のファイル名は変更されない |
| 3 | 0 | ストリングにタイプが入っていない場合，FCB内のタイプは3つのスペースにセット |
| | 1 | ストリングにタイプが入っていない場合，FCB内のタイプは変更されない． |

●I/Oコントロール(ファンクション44H)(注2)

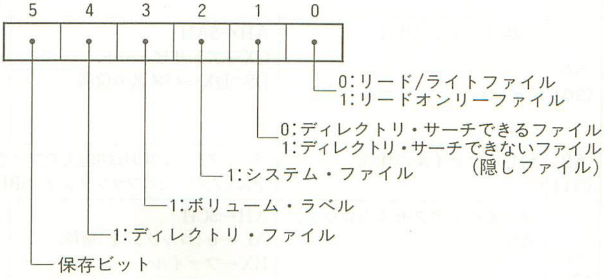
| no. | 機能 |
|---|---|
| 0 | 装置情報をDXに返す |
| 1 | 装置情報のセット |
| 2 | CXで指定されたバイト数のデータをデバイスコントロールチャネルからDS:DXで示される領域へ読む |
| 3 | CXで指定されたバイト数のデータをDS:DXからデバイスコントロールチャネルに書き込む |
| 4 | BL内のドライブ番号に対して2と同様な処理をする |
| 5 | BL内のドライブ番号に対して3と同様な処理をする |
| 6 | 入力ステータスの取得 |
| 7 | 出力ステータスの取得 |

●標準ファイル・ハンドル(注3)

| 番号 | ハンドル |
|---|---|
| 0 | 標準入力 (キーボード) |
| 1 | 標準出力 (CRT) |
| 2 | 標準エラー出力 (CRT) |
| 3 | 標準補助装置 (RS-232Cなど) |
| 4 | 標準プリンタ |

●ファイルの属性(注4)

以下のビットをセット／リセットすることにより属性を指定する．




従って一般のリード／ライト・ファイルの属性は0，リード・オンリーファイルは1，リード／ライト可能な隠しファイルは2，リード・オンリーの隠しファイルは3…と属性が決まる．

## ●MS-DOS ver 3.1における追加ファンクションコール

| Hex | 機　能 | 入力パラメータ | リターン情報 |
|---|---|---|---|
| 1B<br>(27) | ディフォルト・ドライブの諸元を得る | AH←1BH | AL←1クラスタ当りのセクタ数<br>CX←1セクタ当りのバイト数<br>DX←1ドライブ当りのクラスタ数<br>DS:BX←FAT-IDのアドレス |
| 1C<br>(28) | 指定ドライブの諸元を得る | AH←1CH<br>DL←ドライブ番号(0:ディフォルト,1:ドライブA…) | AL←1クラスタ当りのセクタ数(FFHならドライブ指定が無効)<br>CX←セクタ当りのバイト数<br>DX←1ドライブ当りのクラスタ数<br>DS:BX←FAT-IDのアドレス |
| 26<br>(38) | 新しいPSPを作る | AH←26H<br>DX←新しいPSPのセグメント・アドレス | なし |
| 38<br>(56) | 国別情報を得る | ver 2.0とコンパチブル | |
| | 国別情報をセットする | AH←38H<br>DX←FFFFH<br>AL←カントリ・コード | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←2:無効なカントリ・コード |
| 44<br>(68) | デバイスに対するI/Oコントロール | ALに与えるI/Oコントロールファンクションが追加された<br>AL←8 :IOCTLの交換性チェック<br>9 :ドライブのリダイレクト・チェック<br>0A:ファイルのリダイレクト・チェック<br>0B:リトライの回数チェック | |
| 58<br>(88) | アロケーション・ストラテジを得る/セットする | AH←58H<br>AL←0:ストラテジを得る<br>1:ストラテジをセットする<br>BX←0:下位,1:最小,2:上位(AL=1のとき) | キャリーがセットされない場合<br>AX←0:下位, 1:最小, 2:上位(AL=0のとき)<br>キャリーがセットされた場合<br>AX←1:無効なファンクションコード |
| 59<br>(89) | 拡張されたエラーコードを得る | AH←59H<br>BX←0 | AX←エラー・コード<br>BH←エラー・クラス<br>BL←可能な対処コード<br>CH←付加情報コード |
| 5A<br>(90) | 一時ファイルの作成 | AH←5AH<br>CX←アトリビュート<br>DS:DX←パス名の位置 | キャリーがセットされない場合<br>AX←ファイルハンドル<br>キャリーがセットされた場合<br>AX←3:パス名がない<br>5:アクセスできない |
| 5B<br>(91) | 新しいファイルの作成 | ファンクション3CHは指定したファイル名が既に存在すればファイルを空にしたが，このファンクション5BHはエラーを返す点が異る | |
| 5C<br>(92) | ファイル・アクセスのロック/解除 | AH←5CH<br>AL←0:ロック，1:解除<br>BX←ファイル・ハンドル<br>CX:DX←ファイル領域のオフセット | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←1:無効なコード<br>6:無効なハンドル |

| | | | |
|---|---|---|---|
| | | SI:DI←ファイル領域の長さ | 21H:ロックの破壊 |
| 5D (93) | マシン名を得る | AH←5EH<br>AL←0<br>DS:DX←バッファ位置 | キャリーがセットされない場合<br>CX←ローカル・コンピュータ番号<br>キャリーがセットされた場合<br>AX←1:無効なコード |
| | プリンタ・セット・アップ | AH←5EH<br>AL←2<br>BX←インデックス<br>CX←セット・アップ文字列の長さ<br>DS:SI←セット・アップ文字列の位置 | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←1:無効なコード |
| 5F (94) | 割り当てリストのエントリを得る | AH←5FH<br>AL←2<br>BX←インデックス<br>DS:SI←ローカル名のバッファ位置<br>ES:DI←リモート名のバッファ位置 | キャリーがセットされない場合<br>BL←3:プリンタ, 4:ドライブ<br>CX←ユーザ変数域<br>キャリーがセットされた場合<br>AX←1:無効なコード<br>12H:ファイルがない |
| | 割り当てリストのエントリの作成 | AH←5FH<br>AL←3<br>BL←3:プリンタ, 4:ドライブ<br>CX←ユーザ変数域<br>DS:SI←ソース・デバイス名の位置<br>ES:DI←ディスト・デバイス名の位置 | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←1:無効なコード<br>3:パスが見つからない<br>5:アクセスできない<br>8:メモリ不足 |
| | 割り当てリストのエントリのキャンセル | AH←5FH<br>AL←4<br>DS:SI←ソース・デバイス名の位置 | キャリーがセットされない場合<br>→正常<br>キャリーがセットされた場合<br>AX←1:無効なコード<br>0FH:リダイレクトの中止 |
| 62 (98) | 現プロセスのPSP位置を得る | AH←62H | BX←PSPセグメント・アドレス |

# 索　引

## 五十音順

## さ



## た

## 参考文献


『16ビットプログラミング技法』村山仁郎著　産報出版

『ザ・8086ブック』ラッセル・レクター／ジョージ・アレクシー共著　吉川敏則訳　広済堂産報出版

『標準 MS-DOS ハンドブック』アスキー出版局編著　アスキー出版局

『98FAN BOOKS1 CP/M-86 vs MS-DOS』河西朝雄著　技術評論社

『PC-98用拡張システムコール詳説』（1985年インターフェース 4 月号別冊）白田耕作著　CQ出版社

『C プログラミングノート』河西朝雄著　ナツメ社

『NEC PC-9800シリーズ MS-DOS ver 2.0 ユーザーズマニュアル』

『NEC PC-9800シリーズ MS-DOS ver 3.1 ユーザーズマニュアル』日本電気株式会社

河西朝雄（かさい あさお）

昭和49年　山梨大学工学部電子工学科卒
現　　在　長野県岡谷工業高等学校情報技術科教諭
　　　　　第一種情報処理技術者
著　　書　『$N_{88}$-BASIC100％活用法』（技術評論社）
　　　　　『98グラフィックス入門』（技術評論社）
　　　　　『98 FAN BOOKS 1 CP/M-86vsMS-DOS』（技術評論社）
　　　　　『CP/Mによるマシン語（Z80）開発法』（ナツメ社）
　　　　　『Cプログラミング技法』（ナツメ社）
　　　　　『98ソフトウェア技法』（ナツメ社）
　　　　　『構造化BASIC』（技術評論社）
　　　　　『Cプログラミングノート』（ナツメ社）

MS-DOS
実用マクロアセンブラ

昭和61年3月1日　初版　第1刷発行
昭和61年5月25日　初版　第3刷発行

著　者　河西朝雄
発行者　片岡　巖
発行所　株式会社技術評論社
　　　　東京都千代田区九段南2-4-13
　　　　電話　03(262)9351㈹　営業部
　　　　　　　03(262)7671㈹　編集部
印　刷　萬友社
製　本　村上製本所

定価はカバーに表示してあります



ISBN4-87408-814-7 C3055

カバーデザイン＝(株)日本医広




ISBN4-87408-814-7 C3055 ¥2500E　定価2500円